A Survey on Automatic Parameter Tuning for Big Data Processing Systems

Herodotou, Herodotos

2020-04

Herodotou , H , Chen , Y & Lu , J 2020 , ' A Survey on Automatic Parameter Tuning for Big Data Processing Systems ' , ACM Computing Surveys , vol. 53 , no. 2 , 43 , pp. 1-37 . https://doi.org/10.1145/338102

http://hdl.handle.net/10138/318447
https://doi.org/10.1145/3381027

publishedVersion

A Survey on Automatic Parameter Tuning for Big Data Processing Systems

HERODOTOS HERODOTOU, Cyprus University of Technology, Cyprus
YUXING CHEN and JIAHENG LU, University of Helsinki, Finland

Big data processing systems (e.g., Hadoop, Spark, Storm) contain a vast number of configuration parameters controlling parallelism, I/O behavior, memory settings, and compression. Improper parameter settings can cause significant performance degradation and stability issues. However, regular users and even expert administrators grapple with understanding and tuning them to achieve good performance. We investigate existing approaches on parameter tuning for both batch and stream data processing systems and classify them into six categories: rule-based, cost modeling, simulation-based, experiment-driven, machine learning, and adaptive tuning. We summarize the pros and cons of each approach and raise some open research problems for automatic parameter tuning.

CCS Concepts: • **Computer systems organization** → **Self-organizing autonomic computing**; • **Information systems** → *Computing platforms*;

Additional Key Words and Phrases: Parameter tuning, self-tuning, MapReduce, Spark, Storm, stream

ACM Reference format:
Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. 2020. A Survey on Automatic Parameter Tuning for Big Data Processing Systems. *ACM Comput. Surv.* 53, 2, Article 43 (April 2020), 37 pages.
https://doi.org/10.1145/3381027

1 INTRODUCTION

The continuous growth of the World Wide Web, E-commerce, Internet of Things (IoT), and other applications is generating massive amounts of ever-increasing raw data every day [88]. Large-scale data analytics platforms, including both batch data processing systems (e.g., Hadoop MapReduce [4], Spark [6]), as well as stream data processing systems (e.g., Apache Storm [11], Heron [75], Flink [3], Samza [5]), have emerged to assist with the Big Data challenge, i.e., to efficiently collect, process, and analyze massive volumes of heterogeneous data. Achieving good and robust system performance at such a scale is the foundation for successfully performing timely and cost-effective analytics (either offline or in real-time). However, system performance is directly linked to a vast array of configuration parameters, which control various aspects of system execution, ranging from low-level memory settings and thread counts to higher-level decisions such as

Jiaheng Lu and Yuxing Chen are partially supported by Finnish Academy Project 310321.
Authors' addresses: H. Herodotou, Cyprus University of Technology, 30 Arch. Kyprianos Str., Limassol, 3036, Cyprus; email: herodotos.herodotou@cut.ac.cy; Y. Chen, University of Helsinki, C215, Exactum, Kumpula, Gustaf Hallstromin katu 2b, Helsinki, Finland; email: yuxing.chen@helsinki.fi; J. Lu, University of Helsinki, C212, Exactum, Kumpula, Gustaf Hallstromin katu 2b, Helsinki, Finland; email: jiaheng.lu@helsinki.fi.

0360-0300/2020/04-ART43
https://doi.org/10.1145/3381027

resource management and load balancing [88]. Improper settings of configuration parameters are shown to have detrimental effects on the overall system performance and stability [39, 61, 103].

The use of automated parameter tuning techniques is a promising, yet challenging approach for optimizing system performance. The major challenges are:

(1) **Large and complex parameter space:** Hadoop, Spark, and Storm have over 200 configurable parameters each [22, 70, 103]. To make matters worse, some parameters might affect the performance of different jobs in different ways, while certain groups of parameters may have dependent effects (i.e., a good setting for one parameter may depend on the setting of a different parameter) [61, 66].
(2) **System scale and complexity:** As data analytics platforms have grown in scale and complexity, system administrators may need to configure and tune hundreds to thousands of nodes, some equipped with different CPUs, memory, storage media, and network stacks [80]. In addition, executing MapReduce or Spark workloads with iterative stages and tasks in parallel or serial makes it challenging to observe and model workload performance [46].
(3) **Lack of input data statistics:** Data statistics are almost never available for MapReduce and Spark applications, since data often reside in semi- or un-structured files and are opaque until accessed [58]. As for stream applications, the input data are a real-time data stream that typically experiences significant variations in workload properties [34].

Classification of Approaches: A considerable amount of past research tackles the problem of performance optimization by partially or fully automating the process of finding near-optimal parameter values for executing jobs in big data processing systems. This survey performs a comprehensive study of existing parameter-tuning approaches, which address various challenges towards high throughput and resource utilization, fast response time, and cost-effectiveness. Due to the various challenges and scenarios addressed, different strategies or approaches are proposed accordingly. We classify these approaches into the following six categories:

(1) **Rule-based** approaches assist users with tuning some system parameters based on the experience of human experts, online tutorials, or tuning instructions. They usually require no models or log information and are suitable for quickly bootstrapping the system.
(2) **Cost modeling** approaches build efficient performance prediction models by using analytical (white-box) cost functions developed based on a deep understanding of system internals. A few experimental logs and some input statistics are typically required to establish the model.
(3) **Simulation-based** approaches build performance prediction models based on modular or complete system simulation, enabling users to simulate an execution under different parameter settings or cluster resources.
(4) **Experiment-driven** approaches execute an application, i.e., an experiment, repeatedly with different parameter settings, guided by a search algorithm and the feedback provided by the logs of the actual runs.
(5) **Machine learning** approaches establish performance prediction models by employing machine learning methods. They typically consider the complex system as a whole and assume no (or limited) knowledge of system internals (i.e., they treat the system as a black box).
(6) **Adaptive** approaches tune configuration parameters adaptively while an application is running, i.e., they can adjust the parameter settings as the environment changes based on a variety of methods. They enable tuning of ad hoc and long-running applications.

Table 1. Feature Comparison among the Six Parameter Tuning Approaches

Feature	Rule-based	Cost modeling	Simulation	Experiment-driven	Machine learning	Adaptive
Key modeling technique	rules	cost functions	simulation	search algorithms	ML models	mixed
# of parameters modeled	few	some	some	many	many	some
System understanding	strong	strong	strong	light	no	strong
Need for history logs	no	light	light	strong	strong	light
Need for data input stats	no	light	light	no	strong	light
Real tests to run	no	some	no	yes	yes	yes
Time to build model	efficient	efficient	medium	slow	slow	medium
# of metrics predicted	few	few	some	few	many	some
Prediction accuracy	low	medium	medium	medium	high	medium
Adapt to workload	adaptive	light	light	no	no	adaptive
Adapt to system changes	no	no	light	no	adaptive	light

Table 1 summarizes and compares the various key features and functionalities provided by the six approaches in terms of modeling (e.g., technique, number of parameter models, system understanding), need for statistics or runs, prediction accuracy, and adaptability to workload and system changes. It is worth noting that by addressing the number of parameters, system understanding, and need for data input statistics, we summarize the approaches' pertinence concerning the three aforementioned challenges. The six approaches will be analyzed in-depth and compared within the context of both batch and stream data processing systems in the following sections. While most approaches fit neatly into one of the aforementioned categories, some approaches may use techniques that span more than one. For example, Starfish [58, 61] builds detailed cost models for modeling the MapReduce execution process and uses a small simulator component for simulating the task scheduling decisions. In such cases, we categorize the approach based on the predominant technique. Thus, we classify Starfish as a cost modeling approach.

Main Contributions: This manuscript reviews the representative techniques of parameter tuning on big data analytics systems. The comprehensive nature of this survey helps to motivate new parameter-tuning techniques, to develop real-world tuning applications, as well as to serve as a technical reference for selecting and comparing the existing tuning strategies. Particularly, the key contributions of this survey are:

(1) We introduce the area of parameter tuning and provide a general classification of the relevant approaches.
(2) We discuss parameter tuning from various perspectives for two different classes of systems; namely, batch and stream processing systems.
(3) We describe in detail the key features for existing representative tuning methods.
(4) We describe open research problems and showcase that parameter tuning forms a challenging research area, where solutions can be exploited in a variety of real-world use cases.

Related Work: Several existing surveys deal with the performance optimization and/or efficient resource management of large-scale data processing systems. Babu et al. [16] covered the core features of several analytics platforms (e.g., Hadoop, Spark, Dryad), including resource management, task scheduling, and query optimization. In terms of parameter tuning, it briefly discussed only the Starfish line of work [58, 61], elaborated in Section 4.1. Two existing surveys [38, 77] on MapReduce discuss how previous approaches address several of the weaknesses and limitations of MapReduce, including high communication cost, inflexible access to input data, redundant processing, as well as lack of iterations, interactive processing, and support for n-way operations. However, the issue

of configuration parameter tuning is not addressed in either survey. Hirzel et al. [63] present a list of optimizations for stream processing, such as task placement and load balancing. The only relevant optimization discussed is tuning the batching size, which is only one of the many parameters that can impact performance in a streaming setting. Dayarathna et al. [34] focus on the system architecture and use cases of stream processing platforms and only briefly discuss performance scalability. A more recent survey [99] proposes a classification for parallelization and elasticity methods in stream processing systems but lacks any discussion on automatic parameter tuning. Autonomic computing brings together the work of computer science and control communities with the purpose of developing autonomic systems, i.e., systems that can manage themselves automatically. An excellent survey [65] lists the key applications of autonomic computing and discusses the related approaches. Automatic parameter tuning shares a similar aim with autonomic computing in that it improves the big data systems by decreasing human involvement in the parameter configuration process. However, the framework and methods in autonomic computing are quite different from those surveyed here. To the best of our knowledge, no other manuscript exists that deals with parameter tuning to an extent and depth comparable to this survey.

Outline: The remaining manuscript is organized as follows: Section 2 presents background information on the execution and the configuration parameters of popular batch and stream processing systems. Sections 3–8 offer a detailed description of parameter tuning approaches organized into the aforementioned six categories. Section 9 concludes the survey while discussing challenges and open research problems.

2 PRELIMINARIES

Existing large-scale data processing systems can be categorized into *batch* and *stream* processing systems. In the batch model, data are typically collected and stored before being fed into an analytics platform such as Hadoop MapReduce or Spark. In the stream model, however, data arrive continuously and are processed immediately by a streaming platform such as Apache Storm, Flink, or Spark Streaming. In both cases, the execution behavior and performance of the system are controlled by dozens of configuration parameters. Sections 2.1 and 2.2 present the system architecture in conjunction with important parameters for popular batch and stream processing systems, respectively. Finally, Section 2.3 formalizes the problem of automatic parameter tuning.

2.1 Batch Processing Systems

In this survey, we focus on Hadoop MapReduce [4] and Apache Spark [6], the two most popular platforms for big data analytics. *MapReduce* is both a programming model and an execution engine for massive data analysis [35]. As a programming model, MapReduce consists of the *map* (k_1, v_1) and the *reduce* $(k_2, list(v_2))$ user-defined functions. For every input *key-value* pair $\langle k_1, v_1 \rangle$, the *map* (k_1, v_1) function is used to output zero or more intermediate key-value pairs $\langle k_2, v_2 \rangle$, as shown in Figure 1. The intermediate values v_2 are grouped together according to the key values k_2 to generate the pairs $\langle k_2, list(v_2) \rangle$. Next, the *reduce* $(k_2, list(v_2))$ function is invoked for each such pair and produces zero or more output key-value pairs $\langle k_3, v_3 \rangle$. The keys (k_1–k_3) and the values (v_1–v_3) could be of any data type.

A Hadoop MapReduce job executes as a set of parallel map and reduce tasks on a compute cluster (see Figure 1). Each map task processes one *input split*, which is a portion of the input dataset to be processed by the job, and generates a portion of the intermediate data. After map tasks complete, Hadoop uses an external sort-merge algorithm to group all intermediate key-value pairs and *shuffles* (i.e., transfers) them to the cluster nodes that will run the reduce tasks. Finally, the intermediate data will be processed by the reduce tasks, which will generate the final results of the job [123]. The map and reduce tasks can be further decomposed into *phases,* as shown in

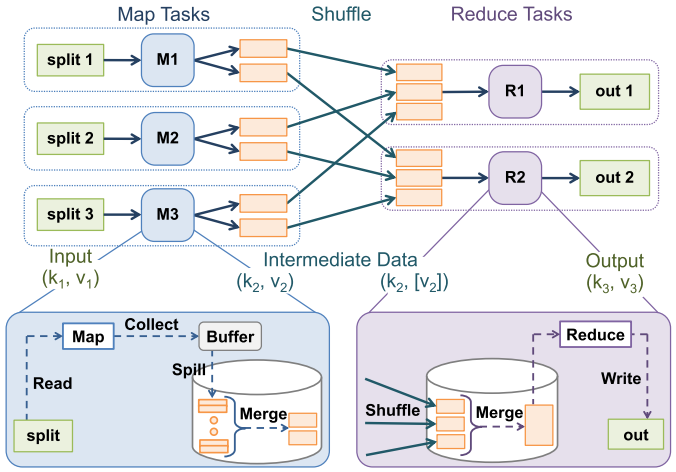

Fig. 1. Execution of a MapReduce job as a set of map and reduce tasks. The map task execution is further decomposed into "read," "map," "collect," "spill," and "merge" phases. The reduce task execution is further decomposed into "shuffle," "merge," "reduce," and "write" phases [58].

Figure 1. The map task execution is composed of five phases: (a) *"read"* for reading the data input from the distributed file system; (b) *"map"* for executing the user-defined map function; (c) *"collect"* for buffering and partitioning map outputs; (d) *"spill"* for sorting, combining, and writing map outputs to local disk; and (e) *"merge"* for merging sorted spill files. The reduce task execution is composed of four phases: (a) *"shuffle"* for copying map output data to the reduce node; (b) *"merge"* for merging sorted map outputs; (c) *"reduce"* for executing the user-defined reduce function; and (d) *"write"* for writing the final output to the distributed file system [58].

In the early versions of Hadoop (up to v1.2.1; 2014), map and reduce tasks were running in a predefined number of map and reduce *slots* per cluster node. However, a MapReduce job typically requires many map slots when it starts, while it needs to reduce slots after the map tasks complete [16]. Hence, the static allocation of resources into map and reduce slots often leads to lowered cluster utilization. In the next versions, named *Apache YARN* [111], Hadoop created two separate functional layers: one for allocating resources to running applications and one for managing the application's life-cycle. The resource allocation model in YARN introduced the notion of *resource containers*, which describe node resources such as CPU and memory. In this model, applications can ask for different container specifications at different points of their execution, giving rise to interesting research problems on how to determine the appropriate amounts of resources for each task as well as how to allocate resources among them [16]. Several follow-up works developed extended resource allocation and scheduling mechanisms for geographically distributed big data processing frameworks based on Hadoop MapReduce and Spark [37].

The Hadoop configuration parameters (in both versions) impact several aspects of job execution at the different phases, such as task concurrency, memory allocation, I/O performance, and network bandwidth usage. Currently, Hadoop has over 200 parameters, from which about 30 can have a substantial effect on job performance [16]. Table 2 lists some of the most influential configuration parameters (valid up to Hadoop v3.2.1), which are used in optimizing Hadoop performance. Default parameter values are either provided by Hadoop or are specified by a system administrator and are used unless the user explicitly specifies parameter values during job submission.

An early comparative study between Hadoop and two parallel database systems revealed that Hadoop was slower by a factor of 3.1 to 6.5 in processing several data-intensive analytical

Table 2. Key Performance-aware Configuration Parameters for Hadoop [48]

Parameter Name	Parameter Description	Default
"dfs.block.size"	The default block size for files stored on HDFS	128 MB
"mapreduce.job.maps"	Number of map tasks	2
"mapreduce.job.reduces"	Number of reduce tasks	1
"mapreduce.combine.class"	A combine function for preaggregating map outputs before the shuffle phase	null
"mapreduce.map.combine.minspills"	Min number of map output spill files present for using the combine function	3
"mapreduce.map.output.compress"	Flag for enabling compression of map output data	FALSE
"mapreduce.map.sort.spill.percent"	Percent of "mapreduce.task.io.sort.mb" buffer to fill before spilling the map output to local disk	0.8
"mapreduce.output.fileoutputformat.compress"	Flag for enabling compression of job output data	FALSE
"mapreduce.reduce.input.buffer.percent"	Percent of reducer's memory devoted to buffering map output data while executing the reduce task	0
"mapreduce.reduce.merge.inmem.threshold"	Max number of shuffled map output pairs before initiating merging during the shuffle	1000
"mapreduce.reduce.shuffle.input.buffer.percent"	Percent of reducer's memory devoted to buffering map output data during the shuffle	0.7
"mapreduce.reduce.shuffle.merge.percent"	Percent of reduce task's memory to fill before initiating merging during the shuffle	0.66
"mapreduce.reduce.shuffle.parallelcopies"	Max number of parallel threads that transfer data from map tasks to a reduce task	5
"mapreduce.task.io.sort.factor"	Max number of data streams to merge during external sorting	10
"mapreduce.task.io.sort.mb"	Size of memory buffer that stores the map output data	100MB
"mapreduce.tasktracker.map.tasks.maximum"	Max number of map tasks executed concurrently at a cluster node (number of map slots)	2
"mapreduce.tasktracker.reduce.tasks.maximum"	Max number of reduce tasks executed concurrently at a cluster node (number of reduce slots)	2

workloads [94]. Motivated by these results, two performance studies [15, 68] conducted in-depth analyses of Hadoop to find the most critical factors and configuration parameters that affect its performance. Both studies concluded that by meticulously tuning these factors and parameters, the performance of Hadoop could be dramatically improved and be more comparable to the performance of parallel database systems.

A follow-up study [126] performed Principal Component Analysis (PCA) to determine the Hadoop configuration parameters that significantly affect system performance, yielding three principal components. The first one is composed of configuration settings that directly affect the I/O performance of MapReduce jobs: `io.sort.factor` and parameters for compressing map and final output. The second component contains settings that control the memory size used to buffer intermediate data (i.e., JVM memory and buffer sizes for intermediate data stored in map and reduce tasks). The final component reflects the parallelism in the MapReduce job and contains the number of tasks and data copiers during the shuffle.

Apache Spark [6] is an open-source distributed framework that simplifies the development of applications that execute in parallel on computing clusters in a fault-tolerant way. Its interface is based on the notion of the *RDD* ("*Resilient Distributed Dataset*"), which is a multiset of objects with a read-only property spread over a cluster and maintained in a fault-tolerant way [132]. As a programming abstraction, an RDD represents a multiset of objects that can be split through a high-performance computing cluster. Operations on RDDs are also divided and executed in

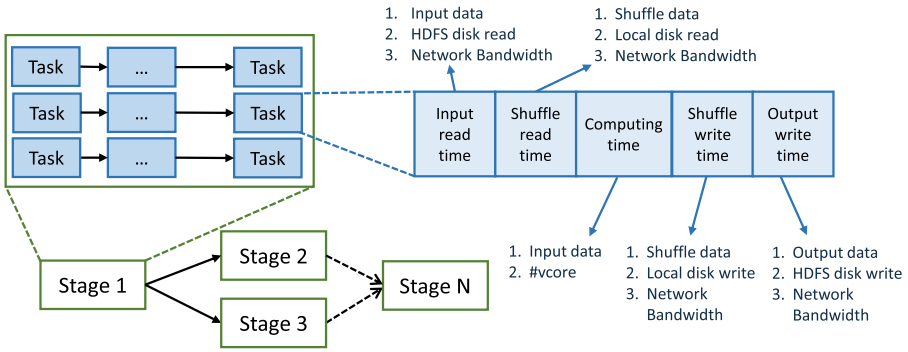

Fig. 2. A Spark job execution consists of multiple parallel or sequential stages and each stage is composed of multiple tasks. The performance of each task is influenced by several factors at each task phase [46, 103].

parallel across the cluster, leading to fast and scalable parallel processing. The overall architecture is composed of the Spark Core and a set of libraries, presented next.

Spark Core: Spark Core is the fundamental framework, providing task distribution, scheduling, as well as basic I/O functionalities. It also exposes an application programming interface (API) for the RDD data objects. The interface involves several operations (e.g., *map*, *filter*, *reduce*) on RDDs by calling a function on Spark in a developer-friendly fashion, concealing the complexity of distributed processing. Figure 2 presents the execution of a Spark job decomposed into stages and tasks [46, 103]. In a specific task, the overall computation time consists of procedures for input read, shuffle read, computing, shuffle write, and output write times. The computation time for each of these procedures may be affected by different factors. For example, computing time may be affected by the input data size and the number of assigned CPU cores.

Spark Libraries: Spark's popularity as a big data analytics platform has grown mainly due to its mature libraries, which are widely used in academia and industry. The four main libraries are:

(1) *Spark SQL* [14], which supports standard SQL connectivity as well as a standard interface for reading from and writing to other data stores including HDFS, Apache Hive, Apache ORC, JDBC, and Apache Parquet, all of which are natively supported.
(2) *Spark MLlib* [90], which is a framework for running machine learning (ML) algorithms distributedly on Spark. Due to its distributed in-memory nature, it can significantly speed up iterative tasks, commonly found in ML applications. Many popular ML and statistical algorithms have been implemented, including regression, classification, and clustering.
(3) *Spark GraphX* [89], which is a distributed graph processing framework on Apache Spark, providing two different APIs for implementing large-scale parallel algorithms: (i) a Pregel abstraction and (ii) a general MapReduce style API.
(4) *Spark Streaming* [7], which employs Spark fast scheduling capability to enable streaming analytics by ingesting and performing RDD transformations on data in a mini-batch fashion. In this way, batch and streaming operations can share the same code and run on the same framework, thus increasing development efficiency.

Spark Configuration: Table 3 lists some of the most common parameters (valid up to Spark v2.4.4) that need to be configured by the users and that can have a remarkable impact on performance. These parameters mainly affect some aspects of execution and allocation of computing resources, including CPU, memory, disk bandwidth, and network utilization.

Table 3. Key Performance-aware Configuration Parameters for Spark Applications [104]

Parameter Name	Brief Description and Use	Default
"spark.default.parallelism"	Number of RDD partitions created by transformations	depends on scheduler
"spark.driver.cores"	Number of cores used by the Spark driver process	1
"spark.driver.maxResultSize"	Max size of serialized results per Spark action	1GB
"spark.driver.memory" (DM)	Memory size for driver process	1GB
"spark.driver.memoryOverhead"	Off-heap memory size per driver	DM*0.10 (min 384 MB)
"spark.executor.cores"	Number of cores for each executor	1
"spark.executor.memory" (EM)	Memory size for each executor process	1 GB
"spark.executor.memoryOverhead"	Off-heap memory size for each executor. It increases with the executor size (often 6-10%)	EM*0.10 (min 384 MB)
"spark.executor.pyspark.memory"	Memory size to PySpark in each executor in MiB	Unlimited
"spark.files.maxPartitionBytes"	Max number of bytes to group into one partition during file reading	128 MB
"spark.memory.fraction"	Fraction for execution and storage memory. It may cause frequent spills or cached data eviction if set too low	0.6
"spark.memory.storageFraction"	Storage memory percent exempt from eviction. When set too high, tasks spill to disk more often	0.5
"spark.reducer.maxSizeInFlight"	Max map outputs to collect concurrently from every reduce task	48 m
"spark.shuffle.compress"	Boolean flag to compress files of map output	true
"spark.shuffle.file.buffer"	In-memory buffer size per shuffle output stream	32 KB
"spark.shuffle.spill.compress"	Boolean flag for compressing data spilled during shuffles	true

2.2 Stream Processing Systems

Distributed stream processing systems are designed to analyze potentially unbounded streams of continuous data in real (or near-real) time [105]. There are two main processing models: *per-record* (or continuous operator streaming) and *microbatching* (or batched streaming) [34]. In the former, applications fetch and process each record individually, providing typically very low latencies, albeit lower throughput compared to microbatching. Apache Storm [11], Heron [75], and Flink [3] are example platforms following the per-record model. With microbatching, the data stream is divided into mini-batches, with each mini-batch processed through the entire application at once. Thus, microbatching can lead to higher throughput but also higher average latencies. Example systems that implement microbatching are Spark Streaming [7] and Storm Trident [13].

Apache Storm [11] is a popular distributed stream processing system that supports real-time computation on clusters of commodity machines. The core unit of data in Storm is a *tuple*, an ordered list of named field values based on a schema. An unbounded sequence of tuples (with the same schema) is called a *stream*. The programming model in Storm entails the creation of a *topology*, a directed acyclic graph (DAG) of spouts and bolts that implement a particular application, as shown in Figure 3. A *spout* is a source of one or more streams. It typically reads tuples from an external source (e.g., web crawler, Kafka topic, sensor) and emits them into the topology. A bolt then receives one or more streams, performs a streaming computation (e.g., filtering, aggregations, joins), and may output one or more streams to downstream bolts. Each spout or bolt can have multiple instantiations as individual tasks that run in parallel. Incoming streams are split and routed among the tasks based on *stream groupings*. For instance, with shuffle grouping, streams are distributed randomly to the bolt's tasks, whereas fields grouping routes a stream based on a subset of its fields. A Storm topology will run continuously on incoming data until it is terminated.

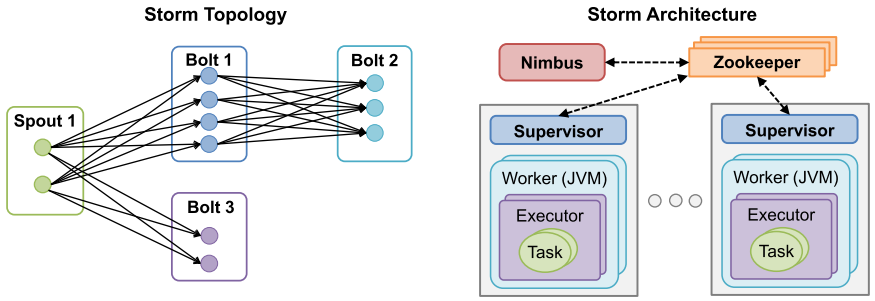

Fig. 3. Storm topology (left) and architecture (right).

Table 4. Key Performance-aware Configuration Parameters for Storm Topologies [12]

Parameter Name	Parameter Description	Default
"supervisor.slots.ports"	Number of Worker processes per machine	4
"topology.workers"	Number of Worker processes for the entire topology	1
"parallelism hint (ph)"	Number of Executor threads per spout or bolt in a topology	1
"topology.tasks"	Number of tasks per spout or bolt in a topology	ph
"topology.max.task.parallelism"	Max number of Executor threads for any spout or bolt	null
"topology.worker.receiver.thread.count"	Number of tuple receiver threads per worker	1
"topology.acker.executors"	Number of Acker threads to spawn for the topology	null
"topology.max.spout.pending"	Max number of tuples to be pending on a spout task	null
"topology.executor.receive.buffer.size"	Size of receive queue per Executor	32,768
"topology.transfer.buffer.size"	Size of outbound message (transfer) queue per Worker	1,024
"topology.producer.batch.size"	Number of tuples to batch before sending to destination Executor	1
"topology.transfer.batch.size"	Number of tuples to batch before sending to destination Worker	1

The Storm architecture depicted in Figure 3 consists of three key components—namely, the state manager (Nimbus), the coordinator (ZooKeeper), and a processing node (Supervisor) running on each worker machine. *Nimbus* is responsible for scheduling tasks to machines as well as handling various types of node and task failures. A *Supervisor* receives work assigned to its machine and creates *Worker* processes. In each Worker, multiple *Executor* threads run the actual spout and bolt *tasks* of a specific topology. Workers also execute the optional "acker" threads for handling tuple acknowledgments and instructing spouts to resend a given tuple in case of failure. Acker threads implement the at-least-once processing guarantees of Storm [109]. Finally, *Zookeeper* handles all communication between the Nimbus and the Supervisors and maintains all their state.

Table 4 lists some of the key configuration parameters (valid up to Storm v2.1.0) that can impact the performance of a Storm topology (i.e., application). Among them, the *degree of parallelism* (*DoP*) has been shown to influence the performance of streaming systems the most [22, 108]. In Storm, the DoP is configured in three ways: (i) a topology is processed by multiple Workers in parallel; (ii) a Worker runs multiple Executor threads in parallel; and (iii) an Executor thread runs multiple Tasks in parallel. Moreover, the number of Executors and Tasks are set individually for each spout and bolt in a topology. Hence, the topology size determines the number of parameters to be tuned.

Heron [75] has been developed and open-sourced by Twitter as a re-implementation and successor of Apache Storm with several architectural improvements to achieve higher throughput and lower latencies. Some of the key changes involve using process-based resource isolation for

better reliability as well as cluster resources on demand for better resource efficiency. The overall data flow, though, remained similar to Storm as described above, while Heron is fully backward compatible with Storm. In addition, Heron introduced rate control using a backpressure mechanism, with which a bolt can slow down upstream operators when it is unable to keep up with the current data rate. This action avoids the accumulation of tuples in long queues, which could have resulted in increased latencies. The existence of backpressure can be used as an indication of performance saturation and, hence, as a signal for updating the current performance models [17, 71] and changing configuration parameters [42] for returning the topology to a healthy state.

Spark Streaming [7] follows the microbatching model, which represents stream computations as a sequence of micro-batch computations on predefined small time intervals. The underlying abstraction is a *discretized stream* (*DStream*), which is represented as a sequence of resilient distributed datasets (RDDs), each containing data from one time interval. Any Spark transformation applied on a DStream translates to transformations on the underlying RDDs, which are continuously executed on each micro-batch and always contain the same set of stages and tasks [74]. Apart from the Spark parameters listed in Table 3, Spark Streaming applications need to set the very important batch interval parameter. A larger batch interval may enable Spark to process data at higher throughput but may also increase the end-to-end latency of processing each record [32].

2.3 Parameter Tuning Problem Statement

A job J executing on a batch or stream data processing system is of the form $J = \langle p, d, r, c \rangle$, where p denotes the program running as part of the job, d the input data properties, r the cluster resources, and c the set of configuration parameter settings used by J. Let c_i denote the ith configuration parameter, taking values from a finite domain $D(c_i)$. The *configuration space* $\mathbb{S}$ is the Cartesian product of the domains, i.e., $\mathbb{S} = D(c_1) \times \cdots \times D(c_n)$ and $c = \langle c_1, \ldots, c_n \rangle$. The performance of job J is portrayed as $perf = F(p, d, r, c)$, where $perf$ is a performance metric of interest such as throughput, latency, or resource efficiency.

The *parameter tuning problem* is defined as follows: Given a program p to process input data d over cluster resources r, find the optimal configuration parameter settings c^* that maximizes F over the configuration space $\mathbb{S}$:

$$c^* = \underset{c \in \mathbb{S}}{\operatorname{argmax}} F(p, d, r, c). \tag{1}$$

The performance function F is usually unknown or partially known from past measurements, while several experiments with Hadoop, Spark, and Storm have shown it to be non-convex and multi-modal [46, 61, 66]. Moreover, finding an optimal solution in such a setting is NP-hard [122]. Despite the many differences between batch and stream processing systems, the parameter tuning problem formulation is identical in both cases. The key difference lies in the performance function F: Batch systems typically focus on optimizing the execution time and/or throughput of jobs, while stream systems focus on end-to-end latency of records or microbatches.

Several past approaches have addressed the general issue of parameter tuning, each trying to resolve one or more of the following specific (sub)problems:

(1) **Avoidance:** to identify and avoid error-prone configuration settings.
(2) **Ranking:** to rank parameters according to the performance impact they exert on the system.
(3) **Profiling:** to collect useful information from previous runs for later prediction and use.
(4) **Prediction:** to predict the workload performance under hypothetical parameter changes.
(5) **Tuning:** to recommend parameter values to achieve objective goals.

3 RULE-BASED APPROACH

Irrespective of the platform used, data analysts and system administrators often struggle with finding proper parameter settings for their applications, so they typically rely upon intuition, experience, data domain knowledge, and advise on best practices from other experts or tuning guides to tune their applications [16]. These rule-based tuning guides range from books and websites to automated systems offering suggestions, as outlined in this section.

3.1 Batch Processing Systems

Hadoop books [123], online tutorials [50, 51], and parameter tuning guides proposed by the industry (e.g., by Hortonworks [91] and Cloudera [31]) offer several *rules of thumb* for setting configuration parameter settings. For example, the Hadoop parameter controlling the number of reduce tasks should be set to approximately 0.95 or 1.75 times the number of reduce slots available in the cluster. The reasoning is to make certain that reduce tasks can execute concurrently but still have slots available for re-executing failed or slow tasks. Other tuning advice requires executing a job first for collecting some information to work effectively. For instance, the average map output size obtained only after execution is needed to set "`io.sort.mb`" effectively.

Various white papers [2, 50, 54] are also helpful to MapReduce non-experts for setting desirable parameter values for their applications. Most of them provide step-by-step guidelines on how to set up a Hadoop MapReduce cluster and manually test settings for various configuration parameters until the desired performance is reached. After a MapReduce job completes execution, post-job performance analysis and diagnostics can be useful in identifying performance bottlenecks. Hadoop Performance Monitoring UI [49] and Hadoop Vaidya [52] parse the job execution log and utilize a set of predefined diagnostic rules to recommend a set of tuning qualitative actions for any performance problems that they identify. For instance, Vaidya will recommend increasing parameter *io.sort.mb* when the number of map spilled records divided by total map output records is larger than a predefined threshold, but without specifying by how much.

The official Spark website hosts a dedicated tuning guide offering best practices on how to tune a Spark application [10]. Parameter tuning guides are also proposed by various industry vendors such as Cloudera [30], Databricks [9], and DZone [8]. Particular emphasis is placed on memory tuning, shuffling, and partition tuning, which are common sources of performance issues.

Memory tuning: Finding good memory parameter settings is crucial for performance due to the in-memory computing nature of Spark. Memory consumption mainly consists of two kinds: *execution* and *storage*. Memory in execution refers to consumption in shuffle, join, sort, and aggregation stages, while memory in storage refers to consumption for caching and propagating input or intermediate data across the cluster. Both categories are designed to share a unified region, M, meaning that storage can acquire all available memory when no execution memory is used and vice versa. During execution, the memory may evict storage when required, but only when there remains a certain threshold R of storage memory available. R specifies a sub-region within the unified region M, where a cached block is never evicted. The two most relevant memory configurations are: (i) `spark.memory.fraction`, which represents the fraction of memory to use for M, set to 0.6 by default. The rest is reserved for data structures, metadata, and safeguarding; (ii) `spark.memory.storageFraction`, which represents the threshold size R set to 0.5 by default.

Shuffle: Shuffles are costly operations, since shuffle data have to be written to disks and then transported across the network. Therefore, avoiding common pitfalls and picking the right arrangement can significantly reduce the number of shuffles and improve an application's performance. *Repartition*, *cogroup*, *join*, and all of the **By* or **ByKey* transformations may lead to shuffles.

These transformations should be considered for use in an appropriate way: (a) *groupByKey* performs an associative reductive operation, which transfers the entire dataset across the network; (b) *reduceByKey* aggregates the value by keys, which computes the aggregate for each key in each partition; (c) *flatMap-join-groupBy* should be avoided—it is better to use *cogroup* when two datasets have been involved in a *groupByKey,* because of the unpacking and repacking of the groups.

Partition tuning: The number of tasks (i.e., the degree of parallelism) in a Spark application is determined based on the number of partitions from input RDD. Tuning the number of the partitions is essential, because setting a good value helps to better utilize the cores available in the cluster and avoids excessive overhead in managing small tasks. For example, if the number of tasks is smaller than the number of slots available to execute them, then CPU resources are wasted.

3.2 Stream Processing Systems

Similar to the case of Hadoop and Spark, books [1], online resources [12], and professional tuning guides [93] offer advice on how to tune Apache Storm and other stream processing systems. As the degree of parallelism is crucial to good performance, there is a particular emphasis on parallelism-related parameters. For example, it is recommended that the number of Executor threads for bolts should be a multiple of the number of worker machines, while for spouts, it should be a factor of the number of partitions of the source [86]. Such settings are expected to improve load balancing in the cluster. Other suggestions include maintaining a ratio of one task per Executor to prevent the context switching overhead among tasks, as well as having one Acker thread per Worker process [41]. Besides, the total number of CPU-bound tasks should not exceed the total number of Workers to avoid CPU contention, while I/O-bound tasks could exceed that limit [1]. However, following such rules requires a deep understanding of the performance characteristics of each spout and bolt in a topology; something that the average data analyst may not possess.

Dhalion [42] is a recent rule-based auto-tuning system for Twitter Heron [75]. A user can define three sets of policies for detecting symptoms, generating a diagnosis, and applying resolution actions. The first policy set consumes system metrics and tries to detect anomalies in the execution of the topology, such as the presence of backpressure or skew. The second set combines the symptoms and attempts to generate a diagnosis that describes the root cause of the problem, such as the existence of a slow process or lack of resources. Finally, a corresponding resolution action is taken to resolve the identified problem, such as increasing the parallelism for a particular bolt. While Dhalion offers a flexible architecture and enables rule-based automation, defining the appropriate policies requires substantial human expertise in performance engineering.

3.3 Discussion

Rule-based approaches are often used in production environments, as they are typically simple to implement, they do not require extra specialized software, and can often be effective in avoiding very bad execution performance [16]. In addition, some parameters are fairly easy to adjust, as they depend on the (fixed) cluster resources, such as setting the number of reducers in Hadoop or the number of Executors in Storm (as discussed above). Using simple but efficient rules for some parameters makes it easy to identify and avoid error-prone configuration settings, as well as to provide tuning towards better performance. However, the overall process of tuning is often time-consuming and labor-intensive, since it typically requires several trial-and-error executions, some of which involve the risk of performance degradation. Finally, most rules are suggested by experts and require in-depth knowledge of system internals to be applied [108]. Overall, rule-based approaches are useful for bootstrapping an application and avoiding bad parameter settings, but are hard to use for obtaining near-optimal performance.

Table 5. An Overview of Cost Modeling Approaches for Optimizing Configuration Parameter Settings

	Approach	Profiling	Prediction	Optimization
Hadoop MapReduce	Starfish [58, 61]	Dynamic instrumentation	Analytical models, black-box models, and simulation	Recursive Random Search
	Predator [121]	Dynamic instrumentation	Analytical models	Grid Hill Climbing
	MRTuner [102]	Information from logs	Producer-Transporter-Consumer analytical model	Grid-based Search
	MR-COF [84]	Dynamic instrumentation	Analytical models/MRPerf [118] simulations	Genetic Algorithm
	ARIA [114]	Information from logs	Analytical models	Lagrange Multipliers
	Zhang et al. [133]	Dynamic instrumentation	Platform- and job-level analytical models	N/A
	HPM [116]	Information from logs	Scaling analytical models and linear regression	Brute-force Search
	IHPM [73]	Information from logs	Scaling models and locally weighted linear regression	Lagrange Multipliers
	CRESP [26]	Information from logs	Analytical models and linear regression	Brute-force Search
	Elastisizer [60]	Dynamic instrumentation	Same as Starfish and M5 regression-tree model	Recursive Random Search
Spark	Wang [120]	Sample job execution	Analytical model	N/A
	Ernest [113]	Sample job execution	Parametric model & NNLS	N/A
	Dione [131]	Sample job execution and Graph Edit Distance	Parametric model & NNLS	N/A
	Singhal [103]	Sample job execution	Analytical models	Grid Search
	DynamiConf [46]	Profile benchmarks	Parametric models	Iterated Local Search
	Chen et al. [28]	Sample job execution	Analytical models	Range Search
Storm	Sax et al. [101]	Performance metrics	Analytical model	Direct Algorithm
	Bedini et al. [20]	Performance metrics	Fine-grained cost model	N/A
Heron	Trevor [17]	Performance metrics	Linear models & Net flow	N/A
	Caladrius [71]	Performance metrics	Timeseries & Cost models	Topological Sorting

4 COST MODELING APPROACH

Cost-based optimization is a well-established technique in database management systems that uses cost functions and data/cost statistics to find a near-optimal query execution plan for an SQL query. However, most of that work cannot be transferred and reused in the big data setting due to the following major challenges: (1) the user-defined functions are developed using high-level programming languages such as Java, Python, or Scala; (2) the absence of data schema and statistics for the input data; and (3) the high-dimensionality of the configuration parameter space [15]. The work presented in this section (and summarized in Table 5) is an attempt to overcome these challenges and use a cost modeling approach for optimizing parameter settings for big data systems.

4.1 Batch Processing Systems

The *Starfish* line of work [57, 58, 61] pioneered cost-based optimization in MapReduce and offered a comprehensive solution for automatically finding good values for the configuration parameters of MapReduce jobs. Starfish proposes a *profile-predict-optimize* approach using a variety of analytical and cost-based models. In particular, a *Profiler* is introduced for collecting detailed statistical information from MapReduce job executions to learn their runtime behavior. The Profiler employs dynamic instrumentation for collecting this information from unmodified MapReduce programs using the BTrace tool [23]. The generated job profile includes detailed dataflow counters and statistics (e.g., number of map output records, average record size) as well as cost counters and statistics (e.g., shuffle execution time, average time to generate a record). Further, a *What-if Engine* [59]

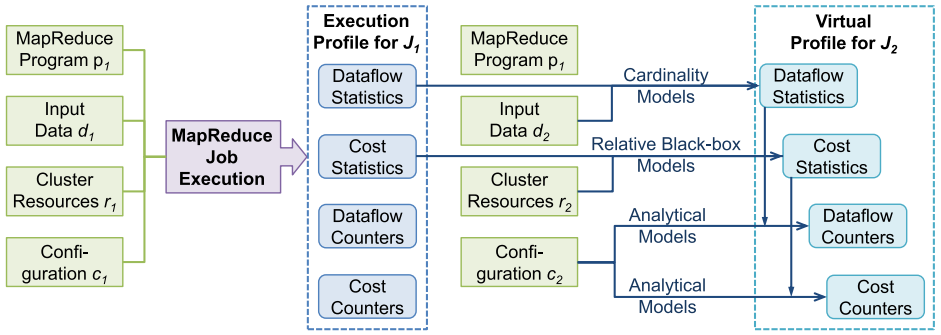

Fig. 4. Overall process used by Starfish to predict a virtual job profile.

deploys appropriate cost models for estimating the effect of hypothetical tuning choices on job performance and generates a virtual job profile, as shown in Figure 4. Specifically, the What-if Engine uses cardinality models from database query optimization for estimating dataflow statistics, relative black-box models for estimating the cost statistics, and analytical models for estimating dataflow and cost counters [56]. Next, a light-weight custom simulator is used along with the virtual job profile to calculate the expected total job execution time. Finally, a *Cost-based Optimizer* uses Recursive Random Search (RSS) [127] for searching the large space of configuration settings efficiently and finding near-optimal MapReduce configuration settings.

Predator [121] is a Hadoop configuration optimizer that follows Starfish's profile-predict-optimize approach but focuses on the optimization step. Similar to Starfish, Predator uses the BTrace tool [23] for collecting detailed information about executed MapReduce jobs, while it implements its own (yet similar) performance models [83]. Unlike Starfish, Predator exploits parameter tuning experience from best practices (recall Section 3.1) to assist the optimization process via narrowing down the search space for some parameters based on cluster resources or input data properties. For example, the values for the parameter "`mapreduce.job.reduces`" (i.e., the number of reduce tasks in a job) are constrained in the set between 0.95 to 1.75 times the number of reduce slots in the cluster. Finally, Predator replaces Starfish's RSS with a Grid Hill Climbing (GHC) algorithm when searching for near-optimal configuration settings, which is better at avoiding local optimum points.

MRTuner [102] is a toolkit for optimizing Hadoop configuration parameters that focuses on the parallel execution of MapReduce tasks. MRTuner uses an execution model to designate the relations among tasks, which are affected by four key factors: (i) the number of map task waves (i.e., number of map tasks divided by number of map slots), (ii) the parameter to compressed map output, (iii) the copy speed in the shuffle phase (affected by number of parallel copy threads and number of reduce tasks), and (iv) the number of reduce task waves (i.e., number of reduce tasks divided by number of reduce slots). These findings motivated the introduction of the *PTC* ("Producer-Transporter-Consumer") cost model to predict the execution time of a MapReduce job. The key intuition of the PTC model is that the Producer (i.e., the generation of map outputs), the Transporter (i.e., the shuffling of map outputs), and the Consumer (i.e., the consumption of map outputs) must be optimized together to improve the utilization of system resources as well as reduce job running time. In addition, MRTuner investigated the complicated relations among 20 performance-sensitive parameters, which facilitated the reduction of the search space. This dimensionality reduction allowed MRTuner to develop a faster grid-based search algorithm for finding optimal parameter settings for the execution of MapReduce jobs.

MR-COF [84] proposes a MapReduce parameter optimization framework that also follows Starfish's profile-predict-optimize approach. MR-COF monitors the runtime behavior of executed MapReduce jobs using the BTrace tool [23]. Next, the framework utilizes a cost-based performance model similar to Starfish and uses MRPerf [118], a Hadoop simulator (discussed in Section 5.1), to predict the MapReduce job performance. Finally, MR-COF replaces Starfish's RRS algorithm with a Genetic Algorithm (GA) that tunes parameters iteratively in an attempt to find near-optimal ones. The main reported benefit of GSA over RRS is the avoidance of falling into local optima.

The *ARIA* framework [114] addresses the problem of estimating and allocating resources to different MapReduce jobs to achieve their service level objectives (SLOs) based on job completion deadlines. In doing so, ARIA will first build a job profile to capture in a compact way the most important performance attributes of the job during its map and reduce phases. Next, ARIA introduces a cost model for estimating the amount of resources needed to complete the job within the requested deadline. The estimates are finally used by a new SLO-based Hadoop scheduler, which decides the ordering of jobs as well as the amount of resources allocated to each job. In essence, ARIA optimizes two key configuration parameter settings (namely, the number of map and reduce tasks) for executing a given MapReduce job by using the *Lagrange Multipliers* method.

In all aforementioned works, profiling is performed for each MapReduce job j separately to build a job profile for j, which is then used for predicting j's completion time under different settings. *Zhang et al.* [133] propose a framework that divided the performance characterization of the Hadoop cluster from the performance properties of different MapReduce jobs. In particular, *Zhang* uses a set of microbenchmarks for measuring the performance of all MapReduce phases other than the user-defined map and reduce functions (recall Figure 1) to generate a *platform profile* of a given Hadoop cluster. Note that the running time of these phases depends only on the amount of processed data and the performance of the underlying cluster. Next, a concise *job profile* for each job can be compiled from the logs of past executions to capture the inherent job properties. Finally, a MapReduce cost model combining the platform and job profiles is derived for estimating the completion time of jobs processing new datasets.

Cloud computing is now growing rapidly as a successful paradigm for hosting MapReduce applications, enabling on-demand elasticity, cost reduction, and pay-as-you-go resources. The ease with which non-expert users can launch a cluster of their preferred size on the cloud has introduced new challenges with regards to parameter tuning. Specifically, users must also select the number and type of VMs to allocate along with platform-specific parameters such as the number of map and reduce slots. *HPM* [116] is one of the first works that addresses the problem of finding the optimal number of map and reduce slots using a cost-based approach. HPM builds a compact job profile by executing a MapReduce job on a small dataset on a small cluster and collecting the Hadoop job logs. Then, by using linear regression and applying a scaling technique, HPM can compute the number of map and reduce slots necessary for processing a large dataset while meeting a certain time deadline. Specifically, HPM will iterate over the range of map slot allocations and find the most appropriate value of reduce slots required to execute and finish the job before the deadline.

IHPM [73] builds on the HPM work by improving the performance model to consider (i) the non-overlapping shuffle phase during map executions and (ii) a varied number of reduce tasks. Instead of linear regression, IHPM employs LWLR ("*Locally Weighted Linear Regression*") for estimating the running time of a MapReduce job. Based on this estimation, the IHPM model utilizes the *Lagrange Multiplier* method for calculating the number of map and reduce slots needed by a job to finish execution before a deadline.

Similar to HPM, *CRESP* [26] also tackles the problem of finding the optimal number of map and reduce slots for executing a given MapReduce job. CRESP proposes a MapReduce job cost model

that takes the form of a weighted linear combination of a set of non-linear functions, which model the association between running time, input data size, and free system resources (i.e., map and reduce slots). The model parameters are learned using linear regression after performing several (25–60) diverse test runs on small clusters and small sample datasets. CRESP then uses the learned cost model to determine the (near) optimal number of map and reduce slots that (a) minimize monetary cost while respecting a job deadline or (b) minimize running time while staying within a monetary budget. Both optimization problems are solved using brute-force search techniques.

Elastisizer [60] extended Starfish by intertwining job-level optimization decisions with resource provisioning decisions, including determining the optimal number and type of VM instances to allocate to meet user requirements. Similar to CRESP, Elastisizer relies on multiple executions of a MapReduce job on a small cluster and small dataset for building an M5 regression-tree model. However, the generated model is only used to predict some cost statistics rather than the overall execution time. Elastisizer combines the M5 tree model with other white-box models and simulation for estimating the full job behavior. In addition, Elastisizer uses Recursive Random Search for simultaneously determining the optimal cluster size and job configuration parameters for executing a MapReduce job within a request time or monetary budget.

As far as Spark is concerned, *Wang et al.* [120] build an analytical model to estimate execution time, memory consumption, and the I/O cost for a Spark job. Specifically, to estimate the overall job performance, they first execute the job on a small cluster using an input data sample and then gather performance measures (e.g., runtime, I/O, and memory cost) for each task. To ensure the collection of useful metrics, the small cluster must contain one node for each node type present in the production cluster, and the sample input data must be large enough so that each CPU core processes one input data block. Finally, the collected metrics are plugged into the analytical models to estimate the job runtime on the production cluster.

Ernest [113] builds performance models on Spark according to the performance of a job J on a small data sample and uses them to predict J's performance on large data and cluster sizes. Specifically, it builds a linear parametric cost model based on four terms: (i) a fixed cost representing the computation time of serial tasks; (ii) a linear term of input data capturing the interplay between the number of data records and the inverse of the number of cluster nodes; (iii) a *log*(machines) term to capture some communication patterns; and (iv) a linear term for machines to capture fixed overheads such as scheduling and serializing tasks. Ernest uses a statistical technique to select the most valuable data points to train a model within a given budget. Finally, it employs NNLS ("*Non-Negative Least Squares*") to obtain the most appropriate model based on the training data.

Dione [131] is a profiling framework that builds performance models similar to Ernest to efficiently estimate the execution time of a Spark job. The key difference from Ernest is that Dione uses a graph similarity technique—namely, Graph Edit Distance—to detect whether a new Spark job j_{new} is similar to an already executed one j_{old}. If a match is found, then Dione will reuse the prediction models built for j_{old} to predict the execution time of j_{new}. Hence, Dione avoids the profiling overheads associated with building a different prediction model for each job.

Singhal et al. [103] propose a technique for predicting the job execution time on a Spark cluster by using statistics collected after running the job on smaller data sizes in a smaller-size cluster. Compared to Ernest, this work builds more complex analytical models to capture the overall job execution behavior, task scheduler delay, task JVM time, and task shuffle time. In addition, the approach employs a simulator that simulates the execution time of each stage based on the parallel execution of tasks. The simulator also enables handling of data skew and node heterogeneity. The prediction model captures the effect of input data size, cluster resource size, and three core parameters: the number of executors, cores for each executor, and memory size for each executor.

DynamiConf [46] proposes parametric models for representing the running time of a job as a function of the degree of partitioning and the number of cluster nodes employed. Unlike the aforementioned approaches, its goal is to configure the degree of partitioning for minimizing resource consumption while bounding any increase in execution time. *DynamiConf* formalizes this problem into a *Flow Activity Allocation* (FAA) problem (proved *NP-hard*). Next, they propose a greedy algorithm for solving it that comes in three variations, as well as an approach using *Iterated Local Search* (ILS), which strikes trade-offs between execution time and resource usage.

Chen et al. [28] propose a cost model for Spark performance predictions, which utilize Monte Carlo (MC) simulation to achieve low-cost training. Specifically, MC uses a small amount of data and resources to make a reliable prediction for larger datasets and clusters, even if some data samples exhibit skewness and runtime deviations, because repeated simulations yield reliable profiling characteristics. Compared to Ernest, this work considers network and disk bounds so that it performs better with I/O-bounded workloads.

4.2 Stream Processing Systems

Sax et al. [101] propose simple analytical models for calculating the optimal batch size along with the optimal degree of parallelism for bolt operators to maximize the throughput of a Storm topology. (Note that batching was implemented by the authors as a library on top of Storm.) The key intuition behind the models is that both of these parameters depend on some measurable properties of the operator (e.g., processing time, output data rate) and the network stack (e.g., network message cost). The proposed algorithm would then use the models to optimize a topology starting from the spouts and moving towards the leaves of the topology graph.

Bedini et al. [20] present comprehensive analytical cost models for Storm that capture (i) data flow costs (i.e., the amount of data transferred between the topology operators and the network cost); (ii) data processing costs (i.e., the execution and I/O cost of the operators); and (iii) system management costs (i.e., the communication overhead between the system components and operators). The models are fairly fine-grained and are demonstrated to provide accurate predictions for latency, throughput, and resource consumption.

Trevor [17] is a model-based tuner built on top of Heron and is composed of a modeling component and an optimizer. The modeling component takes as input runtime metrics data and uses linear models to learn (i) the relation between CPU utilization and input data rate and (ii) the output-to-input ratio for each operator in the topology. Next, Trevor creates and solves a network flow problem containing the topology and the learned models to predict the data rate for each component and the entire topology. The final models and a target data rate are used as input to the optimizer, which outputs an efficient configuration for achieving the target, containing the degree of parallelism for each operator, container dimensions, and container count. The optimizer algorithm is based on the insight that rate matching the operator leads to good configurations and thus uses topological sorting for optimizing critical paths successively.

Caladrius [71] is a performance modeling service for Heron that can predict (i) the future traffic load and (ii) the performance in terms of throughput and backpressure for a stream processing job. For traffic load prediction, Caladrius uses time-series modeling based on additive models that fit non-linear and periodic trends together. Such modeling is robust to missing data, trend shifts, and outliers. However, analytical models are used to predict the topology performance in two key scenarios: (1) given the current configuration, how is the performance affected under potential future traffic loads?; and (2) given the current traffic load, how is the performance affected using a different configuration? The performance is predicted at various granularities ranging from individual tasks to the entire topology.

4.3 Discussion

Cost modeling is a typical white-box technique that uses a set of mathematical formulae for predicting job performance. As shown in Table 5, model parameters are typically calculated from log information or sample job executions, while the various approaches use different analytical models. Such models are computationally very efficient to use for finding proper parameter settings, with tuning times ranging from milliseconds to a few seconds, depending on the optimization algorithm used [46, 61]. Moreover, cost modeling can yield predictions with good accuracy, as the plethora of aforementioned approaches have shown. Overall, all cost-modeling approaches focus on profiling and performance predictions, while most of them also offer automatic tuning (see Table 5). However, it is hard for mathematical formulae to capture the complexity of system internals, especially since all big data analytics systems are distributed in nature and have several moving parts and pluggable components. Hence, models often rely on simplified assumptions such as the proportionality assumption (i.e., the output data size of an operator is proportional to its input size) or that all cluster nodes are homogeneous [58, 80]. Finally, cost-based approaches typically model a particular version of the frameworks, and it is hard to adapt them after changes to the underlying execution engine are made [80].

5 SIMULATION-BASED APPROACH

A simulation-based approach can drastically reduce the time needed to study the behavior of applications under different scenarios. Even though such approaches do not tune configuration parameters automatically, they provide the means to estimate application performance on given configurations. At the same time, a simulator can be used for evaluating workload management, new scheduling algorithms, or resource optimization decisions in distributed environments.

5.1 Batch Processing Systems

MRPerf [118] is a pioneer of the Hadoop simulator that provides a fine-grained MapReduce simulation at the level of task phases (recall Figure 1). MRPerf models both inter-rack and intra-rack network interactions, as well as the processing and I/O times of tasks running on individual nodes. Internally, it employs the widely used *ns-2* network simulator [107] and performs discrete event simulation to model the complicated interactions of multiple factors that influence a MapReduce job execution. The input to the simulator consists of node specifications (e.g., CPU, RAM, disk characteristics), cluster topology (i.e., the arrangement of nodes into racks), data layout (e.g., input size, block size, replication factor), and job description (e.g., number of map and reduce tasks, average record size). This input is then used to simulate job performance and generate a detailed execution trace (at the level of task phases) that includes the job running time, bytes of transferred data, and a timeline of all task phases. Over the years, a set of simplifying assumptions and limitations have been identified such as modeling only one storage device per node and only one output replica, not modeling speculative execution, modeling the task compute time to be proportional to the data size rather than depend on the data content, and only supporting homogeneous environments [87, 115, 118].

Another discrete event MapReduce simulator is *MRSim* [53], which simulates network topology and traffic using GridSim [24], while it models the remaining Hadoop components using the discrete event engine SimJava [64]. Compared to MRPerf, MRSim can provide a more detailed simulation that includes multi-core CPUs, multiple HDDs, as well as the effects of several Hadoop configuration parameters (almost all from Table 2) on job completion times. The input to MRSim consists of (1) a cluster topology file containing the network topology and node specifications; and (2) a job specifications file containing the number of MapReduce tasks, the layout of input data

(e.g., number and size of input splits), cost descriptions (e.g., time to process a map record), and the job configuration parameters. The input information is used by MRSim to estimate various task-level counters and timings of a MapReduce job.

Apache Mumak [92] is a MapReduce simulator that can replay traces generated by Rumen [100], an Apache log processing tool. A Rumen trace contains job-related information describing task durations and processed data (e.g., number of bytes read, number of records written). Unlike MRPerf or MRSim, Mumak cannot simulate the execution of a job using different numbers of map and reduce tasks or for a different cluster size compared to what is captured in the execution trace [58]. Rather, Mumak can only reproduce a previously finished experiment for verifying the Hadoop system design or estimating running time for Map and Reduce tasks when using other schedulers. Finally, it simulates the map and reduce tasks but does not model the shuffling phase.

SimMR [115] is another simulation environment for MapReduce clusters that focuses on simulating the task slot allocations and scheduling decisions for multiple MapReduce jobs. SimMR consists of three interconnected modules: (i) a trace generator for creating a replayable workload based on historical logs or a user-defined workload description; (ii) a discrete event simulator engine for emulating the Hadoop MapReduce execution process; and (iii) a pluggable scheduling policy for making scheduling and provisioning decisions. The simulator engine takes as input the job trace and scheduling policy, replays the trace, and generates an output log that describes the behavior of all map and reduce tasks. Compared to Mumak, SimMR is able to replay both real and synthetic traces as well as model the shuffle phase. However, SimMR does not simulate hardware details of the node clusters (e.g., network transfers or HDDs) as done by MRSim and MRPerf, nor does it model several of the Hadoop configuration parameters.

SimMapReduce [106] has a similar design as MRSim in that it is a discrete event simulator developed over GridSim [24] and SimJava [64], while it shares the same goal as SimMR: to ease the testing of different resource allocation and scheduling policies. The user input consists of node specifications (e.g., CPU speed, memory), network topology, data storage (e.g., input size), job configuration with some basic parameters (e.g., number of map/reduce tasks), and three schedulers (user-, job-, and task-level). Unlike the other simulators, SimMapReduce models file transmission times from the filesystem, which are taken into account as part of the job completion time.

HSim [87] is a Hadoop MapReduce simulator that models several parameters that can impact the behavior of MapReduce jobs, including node parameters (e.g., processors, memory, HDDs), cluster parameters (e.g., network topology, schedulers), and Hadoop system parameters (e.g., JVM settings, `io.sort.factor`, `io.sort.mb`). HSim can be used to investigate the effect of the aforementioned parameters on job performance to study the scalability of the job and tune its performance. HSim seems to be the evolution of MRSim, providing finer-grained modeling of the task phases in a MapReduce job execution, support for more Hadoop configuration parameters, as well as support for two task schedulers—namely, FIFO and FAIR [44].

ABS-YARN [82] is a YARN simulator that models the performance of MapReduce workloads by taking advantaging of a deployed virtualized software for executable modeling, called Real-Time ABS. ABS-YARN models and simulates the resource scheduling process, including the *ResourceManager*, *ApplicationMaster*, and containers. With ABS-YARN, users can configure the cluster size and per-node resource capacity as well as evaluate their deployment decisions under different job workloads, job inter-arrival patterns, and configuration parameters.

5.2 Stream Processing Systems

CEPSim [62] is a simulator that focuses on simulating complex event processing and stream processing in a cloud environment. *CEPSim* introduces a query engine based on the Directed Acyclic Graphs (DAGs) model and develops a simulation algorithm based on the event-set abstraction.

Table 6. A Comparison of the Simulation Support Provided by the Various Simulators

	Simulator	Network Traffic	Hardware Properties	System Execution	System Scheduling	Configuration Parameters
Batch	MRPerf [118]	Yes (ns-2)	Yes	Task sub-phases	No	Only basic ones
	MRSim [53]	Yes (GridSim)	Yes	Task sub-phases	No	Most of Table 2
	Mumak [92]	No	No	Only task level	No	Only basic ones
	SimMR [115]	No	No	Task sub-phases	FIFO, Deadline	Only basic ones
	SimMapReduce [106]	Yes (GridSim)	Yes	Task sub-phases	Yes	Only basic ones
	HSim [87]	Yes (GridSim)	Yes	Task sub-phases	FIFO, FAIR	Most of Table 2
	ABS-YARN [82]	No	No	Task sub-phases	FIFO	Only basic ones
Stream	CEPSim [62]	No	Yes	Task sub-phases	Yes	Only basic ones
	Requeno et al. [98]	No	No	Task Level	Yes	Many from Table 4
	Kroß et al. [74]	No	No	Task Level	Yes	Only basic ones
	SSP [81]	No	No	Task Level	FIFO	Only basic ones

The design principle of *CEPSim* consists of generality, extensibility, multi-cloud, and reusability. *CEPSim* can be customized so that it can be applied to different operator placements, different task schedulers, as well as multiple types of cloud environments.

Requeno et al. [98] propose a formal method to simulate the performance of Apache Storm applications. Specifically, they introduce a modeling approach using UML to capture the performance characteristics of the Storm architecture. Next, they apply transformation patterns to convert the UML to performance models using Generalized Stochastic Petri Nets. The final models are then used in an event-driven simulation for predicting throughput and latency.

Kroß et al. [74] propose a general performance model to simulate both batch and streaming processing on Spark. Unlike previous approaches that predict response time, they also model resource utilization. Specifically, they extended the Palladio Component Model (a formal modeling framework to describe performance factors in software) to simulate parallel operations as well as the cluster resources on top of the general simulation platform SimuCom.

SSP [81] is a configurable and executable model of Spark Streaming written in ABS, a formal executable language for modeling distributed systems and virtualized software. SSP models not only the Spark stream processing framework itself, but also stream workloads. Through simulations, users are able to evaluate various deployment decisions and to predict reliable performance metrics such as processing and scheduling delay times.

5.3 Discussion

Performance evaluation of a system is critical for users to achieve good performance for workloads. When users lack information on the characteristic of data, workloads, framework, and/or resources, a simulator enables them to evaluate performance in a relatively safe simulation environment, as well as to make resource or cost decisions for a production environment. The techniques presented in this section are summarized in Table 6. Most of the existing simulators tackle task sub-phases but only involve some basic configurations. Support for network traffic, hardware properties, and scheduling varies across the simulators. Overall, the simulation helps to accurately learn characteristics with a reasonably low training overhead. Hence, simulation-based approaches target the avoidance and prediction aspects of the parameter tuning problem (recall Section 2.3). However, increasing the involved parameters may degrade the simulation quality, while building a simulator requires comprehensively understanding the internal system dynamics as well as data and workloads.

6 EXPERIMENT-DRIVEN APPROACH

Experiment-driven approaches for parameter tuning are typically motivated by the inability of cost models to accurately predict the performance of different applications given their black-box nature and the presence of clusters containing different CPU, memory, storage media, and network stacks [80]. An experiment-driven approach repeatedly executes an application on a cluster with different settings each time, until it converges to some proper settings. So, the critical challenge of these approaches is to reduce experimental runs required to achieve the desired performance. While experiment-driven approaches may yield better parameter settings, they typically take much longer due to the repeated actual executions of applications.

6.1 Batch Processing Systems

Panacea [85] is a compiler-guided software that combines static analysis with trace analysis to perform two types of optimizations for MapReduce jobs. The first one involves bytecode transformations to reduce overheads of iterative applications while the other one tunes job configuration parameters. For parameter tuning optimization, Panacea employs four main components: (1) a *Trace Instrumenter* that modifies application bytecode to include selective trace generation; (2) a *Parameter Instrumenter* that adds statements in the bytecode for setting parameter values; (3) an *Application Executer* that executes the application on sample input datasets and measures the execution time; and (4) a *Parameter Tuner* that defines the search space and invokes the other three components iteratively for different parameter values. The settings that yield the lowest execution time are then coded into the MapReduce job. In an attempt to lower the high dimensionality of the search space, Panacea identifies subsets of parameters that can be searched independently and achieves search times in the order of a few hours [85].

Gunther [80] is a search-based tuning system for Hadoop, which treats automatic parameter tuning as a black-box optimization problem that can be answered using a search algorithm. In particular, Gunther employs a *Genetic Algorithm* that iteratively selects different configuration settings and evaluates their impact on the MapReduce job performance via test runs. The search terminates after a convergence criterion is reached or a certain number of trials are performed. According to the authors, Gunther can find near-optimal parameter settings with less than 30 trial runs. Also, Gunther includes a methodology for lowering the search space dimensionality, which ignores parameters with little effect on performance. Their approach involves using job counters to categorize MapReduce jobs into four categories that are affected by the same or almost the same subset of parameters. After a job is classified into one of the four groups, Gunther limits the search to the group's corresponding subset of parameters for reducing the search time.

Petridis et al. [95] presents a *trial-and-error* methodology for tuning Spark parameters based on the performance of a small number of experiments. Specifically, based on documentation and past execution experience, the authors first select 12 important performance-aware parameters, then test the performance impact of these parameters on three benchmark applications (namely, *sort-by-key*, *shuffling*, and *k-means*). Based on these results, they develop a methodology in the form of a block diagram containing the seven most impactful parameters. Parameter tuning involves executing a Spark application using different settings derived from following the block diagram.

BestConfig [134] is a general tool that can search for proper parameters within a deadline for several big data platforms such as Spark, Hadoop, and Hive. To overcome the high-dimensionality challenge of parameters, *BestConfig* first uses divide and diverge sampling to (i) guarantee a broad coverage of parameters by covering most of the area regions and (ii) pick representative samples from regions instead of using all samples. Next, *BestConfig* deploys a recursive bound-and-search algorithm to explore the possible optimal configuration among the space. *BestConfig* also provides a theoretical explanation of the advantages of combining their sampling and search algorithms.

Gounaris et al. [47] presents a systematic tuning methodology divided into an offline and an online phase. In the offline phase, the authors propose selecting three benchmark applications (i.e., "*sort-by-key*," "*shuffling*," and "*k-means*") and executing them using different configuration settings. The settings are generated by varying each parameter separately and in pairs, generating 117 runs (using 15 parameters) for each application. The collected results are analyzed using a graph algorithm to compose nine complex candidate configurations. In the online phase, a new Spark application is executed nine times using the candidate configurations to determine the best one.

AutoTune [18] is a parameter tuning tool for Hadoop and Spark. *AutoTune* executes experiments on both the production cluster and on a smaller-scale testbed to perform more test runs. The overall approach consists of three main steps. First, *AutoTune* executes the application using different data sizes and numbers of machines to build a parametric model similar to Ernest [113]. The model is used to select the data and cluster size for running the experiments within a time budget. The second step involves (i) exploration via using *latin hypercube sampling* (LHS) for generating different configurations to execute and (ii) exploitation via building and employing a *Random Forest* model for finding promising configurations to execute. In the final step, the best q configurations are used for executing the application on the production and determining the best one.

Yu et al. [129] introduce an auto-tuning method for in-memory data processing frameworks (e.g., Spark) that considers data size and high dimensional parameters. They first propose a hierarchical predictive model, which consists of many sub-models arranged in a hierarchy. The central idea of a hierarchical model is to build several simpler models rather than a single complex model. Next, they employ a Genetic Algorithm to guide the experimental search for the optimal configuration.

6.2 Stream Processing Systems

BO4CO [66] is an experimental-driven method for finding optimal configurations from a large number of parameters that can influence the performance for stream processing systems. Specifically, *BO4CO* bootstraps the optimization using Latin Hypercube Design to choose a representative configuration from the whole space. Then, *BO4CO* employs Gaussian Processes (GPs) to iteratively predict the average and confidence interval of a response variable at unknown configuration areas. For better exploration, *BO4CO* takes advantage of Bayesian networks to make use of previous information of historical logs, and kernel estimators to better seek optimal configuration.

Fischer et al. [41] proposed a method to automatically choose configuration for distributed stream processing. In particular, they leverage a Bayesian network to predict the parameters with good performance. Bayesian Optimization is an iterative process for repeatedly estimating an objective function. In each iteration, the next configuration is predicted based on the known prior historical information. Bayesian Optimization also supports pausing and resuming the processes so that a decision can be made whether to proceed or terminate the tuning.

Bilal et al. [22] introduce a parameter tuning framework for stream processing systems. This framework combines a black-box prediction technique and a rule-based optimization method. They initialize the configuration by Latin Hypercube Sampling (LHS), then leverage a Hill Climbing Algorithm (HCA) to explore the configuration space. The stages in HCA consist of initial sampling, local search, potential best configurations, shrink, and restart. To further speed up the search, the authors heuristically rank the parameters and provide feedback for the configuration.

Liu et al. [86] introduce a stepwise profiling method to tune workload performance on stream processing platforms. To capture the workload characteristics and link it to the resources, they leverage a profiling approach that auto-scales the provisioned resources. The designed paradigm of profiling is a trial-and-error method, which uses feedback flow to iteratively improve the scalability and applicability. The profiling further enables users to make cost-effective or efficient performance decisions for resource provisioning.

6.3 Discussion

Experiment-driven approaches enable the exploration of unknown configuration areas and work independently of system versions or hardware configurations. The knowledge gained from real test runs on real systems helps to further understand the performance implications of data, workloads, systems, and resources. The trial-and-error strategy enables identifying and avoiding error-prone configuration settings, while the experimental results empower profiling and tuning. Generally, most of the methods used in experiment-driven approaches are search-based algorithms, which enable global and local search for the optimal configuration. One primary goal is to reduce the number of experimental runs but still cover all critical areas of the configuration space. However, it remains challenging to find an optimal configuration among such a large configuration space. The repeated experiments lead to expensive resource consumption and time delays, while the test results may lose their effectiveness for dynamic workloads.

7 MACHINE LEARNING APPROACH

When analytical models cannot meet the complexity of systems, jobs, or data, researchers explore alternative black-box techniques to model performance. The typical techniques are machine learning (ML) models, which offer two main benefits: (i) there is no need to understand the internals of the systems, jobs, and data; and (ii) the model itself improves with more training data. However, ML approaches also face two core challenges related to parameter and model selection.

Parameter selection: Even though there exist over 200 configuration parameters in Apache Spark [6], not all of them significantly influence the performance of Spark jobs. Hence, it is crucial to identify the most impactful parameters that control resource allocations (e.g., CPU, disk, memory), job scheduling (e.g., shuffle, parallelism), and data (e.g., compression, task partition). The common way is to select parameters based on experts' experience [119], industry guides (e.g., Databricks [33]), or based on experimental runs. A smarter way is to use machine learning techniques to rank or identify the important parameters that have a strong correlation with performance. Factor Analysis [110] and Principal Component Analysis (PCA) [126] have been used in the past for this purpose.

Model selection: There exist several ML methods to fit the tuning context. Some common ones are: (i) *Decision Trees*, which use a tree-like model of options to decide which parameters might influence performance negatively or positively; (ii) *Logistic or Tree-based Regression*, which measures the relationship between the dependent variables (i.e., parameters) and one or more independent variables such as execution time; and (iii) *Artificial Neural Networks*, which form a collection of connected nodes that can transmit signals to each other, leading to some prediction [55, 119].

Several research efforts have focused on employing machine learning–based performance modeling and tuning, listed in Table 7. We describe the details of this work below.

7.1 Batch Processing Systems

One of the earliest ML approaches is based on insights gained after analyzing 10 months of MapReduce workload traces from a 400-node Hadoop production cluster at Yahoo! [72]. The proposed algorithm follows an instance-based learning approach composed of two steps: (i) identifying a set of comparable recent jobs using *K-Nearest-Neighbors* and (ii) predicting the runtime of a new MapReduce job using regression models and the completion times of similar jobs. For the first step, the authors used the *Heterogeneous Euclidean Overlap Metric* as a similarity measure for jobs based on seven input features: job submission time, job name, user name, number of map/reduce tasks, and map input bytes/records [72]. For the latter step, they used *locally weighted linear regression* to model job running time as a multi-variate linear function of only four input features: the number of map/reduce tasks and map input bytes/records. The key limitation here is the inability to model

Table 7. Machine Learning Techniques, Their Input Features, and Models for Predicting Job Performance

Approach	Input Features	ML Technique
Kavulya et al. [72]	Job submission time, job name, user name, number of map/reduce tasks, and map input bytes/records	K-Nearest-Neighbors
	Number of map/reduce tasks and map input bytes/records and weights based on job similarity	Locally weighted Linear Regression
Kadirvel et al. [70]	Number of nodes, map input size, number of reduce tasks, I/O sort record percent, and the shuffle parallel copies	Gaussian Processes, Multilayer Perception, Discretization regressor, M5 Pruned Model Tree
Yigitbasi et al. [128]	`mapreduce.task.io.sort.mb`, number of map and reduce slots, number of reduce tasks, and map input size	Support Vector Regression
AROMA [76]	CPU, disk, and network utilization patterns	k-mediod Clustering
	Input data size, resource allocations, and several parameters	Support Vector Machine
PPABS [124]	CPU, memory, and disk utilization statistics	k-means++ Clustering
	CPU, memory, and disk utilization statistics	Pattern Recognition
PStorM [40]	Features from code analysis (e.g., map class name) and execution profile (e.g., map size selectivity)	Gradient Boosted Regression Trees
Chen et al. [25]	A set of several configuration parameter values	Tree-based Regression
RFHOC [21]	A set of several configuration parameter values	Random-Forest Approach
Wang et al. [119]	CPU cores, memory, max map output, compress, shuffle, block size, parallelism, input data size, etc.	Decision tree (C5.0), Recursive Random Search
Hernández et al. [55]	CPU cores, memory, parallelism	Gradient Boosting Regressors
PBDST [67]	Simultaneous multi-threading (SMT) configurations	k-means, logistic regression
d-Simplexed [27]	CPU, memory, data size	Delaunay Triangulation
Zacheilas et al. [130]	A set of several configuration parameter values	Gaussian Processes
Li et al. [79]	Memory sizes and number of cores & threads in various stages	Support Vector Regression
Trotter et al. [108]	Number of worker processes, number of executors	Genetic Algorithm, Bayesian Optimization
Trotter et al. [109]	Workers, spout parallelism, bolt parallelism, acker parallelism	Genetic Algorithm, Support Vector Machines
OrientStream [117]	Variety of data-, plan-, operator-, and cluster-level features	Ensemble/Incremental ML
Vaquero et al. [110]	Parameters/metrics selected based on Factor Analysis	Reinforcement Learning

the different types of processing performed by different jobs, even though they might have the same number of tasks and consume similar amounts of input data.

Another work performed an empirical evaluation of how several supervised machine learning (ML) approaches can predict the completion time of MapReduce jobs [70]. The overall approach is decomposed into three steps: (1) identifying a set of factors that can influence job performance; (2) selecting an appropriate set of features to use by establishing which factors have the best predictive capability; and (3) predicting the job execution time using an ML technique. The first step was done via empirically evaluating multiple MapReduce jobs. The factors were organized into six categories: resources (e.g., number of nodes), data (e.g., input size), program (e.g., CPU or I/O intensive), configuration (e.g., number of MR tasks), faults (e.g., number of node failures), and environment (e.g., disturbance from co-located virtual machines). From the feature selection step, the most important factors identified were the number of nodes, map input size, number of reduce tasks, I/O sort record percent, and the shuffle parallel copies [70]. Finally, the best approaches were *Gaussian Process Regression*, *Multilayer Perceptron*, *Regression by Discretization*, and *M5 Pruned Model Tree*, all with a mean percentage prediction error ranging between 4.4% and 11.9%.

A later work investigated several ML-based performance models for predicting MapReduce job execution time and concluded that an SVR ("*Support Vector Regression*") model has good computational performance as well as good accuracy [128]. Instead of performing any feature selection process, the authors relied on their domain expertise and selected five parameters to use as input to the models: `io.sort.mb`, number of map and reduce slots, number of reduce tasks, and map input size. The training data collection is performed separately for each application and involves a sparse sampling approach. For selecting the best parameter settings to use for a MapReduce job, the authors currently perform an exhaustive search over the performance surface produced by the SVR model (even though they recognize the need for using smarter search algorithms such as a Recursive Random Sampling or a Genetic Algorithm for searching) [128].

AROMA [76] is a job provisioning system for finding proper resource allocation (i.e., number and type of VMs) and Hadoop configuration parameters for executing a MapReduce job to achieve a service-level objective. MapReduce jobs exhibiting similar performance behavior were found to have comparable resource consumption characteristics. Driven by this observation, AROMA employs a two-phase approach. In the first offline phase, the jobs are grouped using the *k-mediod clustering algorithm* along with the *Longest Common Subsequence* (LCSS) distance metric. The jobs within each cluster exhibit a similar CPU, disk, and network utilization behaviors. An SVM ("*Support Vector Machine*") model is then trained for each cluster to predict the job's performance given input data sizes, several configuration parameters, and resource allocation. Note that AROMA also applies a stepwise regression approach for detecting the set of features that are the leading predictors for the job performance. In the second online phase, AROMA executes a submitted job on a small cluster with sample input data to obtain its resource utilization pattern, which in turn is used to find the job's cluster and corresponding SVM model. Finally, a pattern search-based optimization approach is applied to the selected SVM model for determining near-optimal configuration parameters and resource allocations that minimize resource cost while respecting the job's execution deadline.

The *PPABS* ("Profiling and Performance Analysis-based System") framework utilizes k-means++ clustering and Simulated Annealing for finding near-optimal configuration settings for Hadoop jobs [124]. Similar to AROMA, it consists of an offline and an online phase, performed by the *Analyzer* and *Recognizer*, respectively. The Analyzer executes various MapReduce jobs under different settings and collects CPU, memory, and disk utilization statistics. By using these attributes, the jobs are classified into a set of equivalence classes using an adapted *k-means++ clustering* algorithm. For each equivalence class, the problem of finding optimal configuration settings is formulated as a Combinatorial Optimization problem, and *Simulated Annealing* is used for finding a near-optimal solution. In the online phase, the Recognizer executes an incoming job on a small input sample to collect CPU, memory, and disk utilization statistics and then applies pattern recognition techniques to categorize it under one of the equivalence classes. Finally, the job is executed using the configuration settings that correspond to the identified equivalence class.

PStorM ("Profile Store and Matcher") [40] is a system that can create execution profiles for newly submitted MapReduce jobs by utilizing information gathered from previous executions of other jobs. For each submitted job, PStorM runs a sample job with one map task and multiple reduce tasks to collect the job's profile using the Starfish Profiler (recall Section 4.1). This profile is used to build a feature vector containing a set of static features obtained by analyzing the code (e.g., map class name) and a set of dynamic features from the execution profile (e.g., map size selectivity). Next, the profile store is searched using the feature vector and *Gradient Boosted Regression Trees* in an attempt to find a matching job profile. If one exists, then the Starfish Cost-based Optimizer will use it to search for optimal parameter settings. Otherwise, the Starfish Profiler is employed to execute the job, collect its profile, and save it in the profile store for future use.

Another work [25] proposes a *tree-based regression* approach consisting of a prediction and an optimization phase. The former one estimates the execution time of a MapReduce job by building three prediction models. The first two models make predictions about the behavior of the map and reduce tasks, respectively. In particular, each model takes as input a set of configuration parameter values and outputs three task features—namely, the median task execution time, the deviation of the task execution times, and the number of task waves. The predicted task features are then used by the third model to predict the overall job completion time. Training data are collected for each MapReduce job by executing it multiple times with randomly generated parameter settings. Finally, the optimization phase searches the space of configuration parameters using *Random Hill Climbing (RHC)*, a combination of Random Sampling and Hill Climbing.

RFHOC [21] proposes a *Random-forest Approach* to tune the parameters of Hadoop jobs that are repeatedly run and/or have long duration. Specifically, RFHOC uses random forest learning to train an ensemble model for each MapReduce task phase (recall Figure 1). The input for each model is a set of several Hadoop parameters (see Table 2), while the output is a performance prediction for that phase. The sum of these predictions serves as an approximation for the total running time. To generate the relevant training data, a MapReduce job is repeatedly run 2K times using different (random) configuration settings each time. Finally, RFHOC utilizes the generated models along with a *Genetic Algorithm* for searching the configuration space of Hadoop.

Wang et al. [119] present a method for tuning Spark parameters using *binary* and *multi-classification*. In the binary classification step, a model is trained and used to predict whether a set of parameters will improve performance. For the cases where the answer is yes, a multi-classification model is trained and used to predict by how much the performance will improve (e.g., by 5%, 10%). This article investigates several common binary and multi-classification algorithms and finds the decision tree model (C5.0) to work best. Finally, the authors use *Recursive Random Search* (RRS) [127] for enumerating the space of possible options and finding the best one.

Hernández et al. [55] utilize a machine learning model for predicting the completion time of Spark workloads based on a set of metrics at both system and application level. In particular, the features used are related to parallelism (e.g., Executor cores and memory), application (e.g., number of tasks, bytes shuffled), and system (e.g., CPU load). The authors experimented with several regression models such as Bayesian Ridge, Linear Regression, SGD Regressor, Lasso, and so on, and concluded that *Gradient Boosting Regressor* has the best accuracy. The prediction model is then leveraged by a heuristic algorithm to recommend near-optimal parameters for task parallelism.

PBDST [67] is a framework for tuning Simultaneous MultiThreading (SMT) configurations (i.e., the thread count in SMT cores) on POWER8 processor architectures for Spark-based workloads. Specifically, PBDST consists of an offline component that is responsible for training the models and an online component that is responsible for predicting a near-optimal SMT configuration in each stage of a Spark application to achieve better performance. The training data are collected using microarchitecture-level profiling and are used to train a classifier for predicting SMT configurations. PBDST implemented several predictors and found that *K-Nearest Neighbor* and *Logistic Regression* led to lower prediction errors.

d-Simplexed [27] is a performance prediction framework for tuning Spark workloads. Unlike other black-box ML methods, *d*-Simplexed leverages a *d*-dimensional mesh using Delaunay Triangulation over a selected set of *d* parameters. The key idea is that piece-wise linear regression models could be built faster and yield better prediction than complex models such as Gaussian processing or Neural Networks. To further speed up model construction time, *d*-Simplexed proposes an adaptive sampling technique to collect as few training points as required for accurate prediction.

7.2 Stream Processing Systems

Zacheilas et al. [130] focus on the proactive adjustment of operator parallelism in stream processing systems. Specifically, they employ Gaussian Processes for predicting the future load and estimated latency based on historical data. These estimations are then used to construct a state transition graph that models the impact of actions (e.g., changing the degree of parallelism) to the system performance. Finally, the graph is utilized for selecting the appropriate transition actions that minimize a cost function and ensures maximum performance while the resources are neither over-utilized nor over-assigned.

Li et al. [79] aims to not only model the workload performance but also offer a scheduling solution for Storm. First, they use a non-parametric regression method called Support Vector Regression (SVR) to model the performance surface of a given workload. The modeling part involves estimating the mean tuple processing latency of each task and the mean tuple transfer latency among tasks. Then, based on the modeled surface and predicted performance, they heuristically search for a scheduling plan that includes the best degree of parallelism for each operator.

Trotter et al. [108] propose to accelerate the exploration of the configuration space in Storm stream processing via employing either Genetic Algorithms or Bayesian Optimization. The system includes two components. The first one, the sensors, collect the relevant performance metrics from Storm and the Java Virtual Machines. The second one, the optimizers, takes advantage of the collected profiles to search the configuration space and find a good configuration point. In follow-up work, *Trotter et al.* [109] refine their Genetic Algorithm and employ a classifier to reduce the search time over the configuration space. In particular, they use Support Vector Machines to filter out candidate configurations that are less likely to produce a good result.

OrientStream [117] is a framework that uses incremental machine learning approaches for modeling and predicting the resource usage (i.e., CPU, memory, latency, and throughput) of workloads in distributed stream processing systems. Specifically, *OrientStream* uses an ensemble of four incremental learning models—namely, Naive Bayes, HoeffdingTree, Online bagging, and Nearest neighbors, which are trained using a variety of data-, plan-, operator-, and cluster-level features. To further improve the training process, *OrientStream* detects and discards abnormal training data. Finally, *OrientStream* can automatically adjust operator parallelism and obtain better performance.

Vaquero et al. [110] apply reinforcement learning to auto-tune Spark Streaming workloads. The authors first use a range of synthetic and real workloads to generate training data. Next, Factor Analysis is used for identifying the most relevant metrics and and k-means for clustering them into meaningful groups. Lasso path analysis is then used to generate a ranked list of parameters based on the impact on latency. Finally, given live metrics and the current configuration, a reinforcement learning module explores the configuration space and selects new configurations iteratively.

7.3 Discussion

We summarize the aforementioned ML-based approaches in Table 7. These papers propose to use different types of ML techniques to predict the performance of several or all parameters by using historical logs or execution metrics as training data. In this context, parameters and metrics are typically treated the same way, all feeding into the models. By leveraging ML methods, users treat the problem as a black-box that is independent of system internals and hardware, in spite of the complexity and scale of data, workloads, systems, and resources. Most of the ML-based approaches focus on building predictive models and tuning performance based on them. During model training, the performance-sensitive parameters and essential metrics can be ranked automatically. With increasing training data sizes, the models can achieve high prediction accuracy and near-optimal recommendations. However, obtaining training data can be a costly process, as it requires runs

Table 8. The Unique Features and Methodology of Adaptive Approaches for Parameter Tuning

	Approach	Unique Features	Methodology
Batch	Polo et al. [97]	Adjusts number of allocated map/reduce slots	Cost functions
	MROnline [78]	Supports aggressive and conservative tuning	Gray-box Hill Climbing
	Ant [29]	Supports heterogeneous cluster nodes	Genetic Algorithm
	JellyFish [36]	Performs dimensionality reduction of search space	Model-based Hill Climbing
	KERMIT [45]	Optimizes CPU and memory for workloads	Global and local search
Stream	T-Storm [125]	Optimizes number of Worker processes in Storm	Traffic-aware Scheduling
	Das et al. [32]	Adapts batch size in Spark Streaming	Fixed-Point Iteration
	DRS [43]	Adapts # of Workers and parallelism hint per operator	Queueing Theory
	Drizzle [112]	Adapts batch size and grouping in Spark Streaming	Group and pre-scheduling
	Petrov et al. [96]	Decides # of AWS worker nodes and Spark Executors	Cost performance model

under different settings to avoid under-fitting. The situation becomes worse as workloads can change dynamically, and unseen applications appear.

8 ADAPTIVE APPROACH

All aforementioned approaches for parameter tuning require information from past job executions to predict and optimize the performance of a workload. However, past executions might not be available or may be inaccurate (e.g., the job was executed over a different dataset), while running the job is computationally expensive. The adaptive approaches discussed in this section and listed in Table 8 avoid extra job executions by tracking the execution of a currently running job and changing its configuration in an online fashion to dynamically improve performance or meet a predefined deadline.

8.1 Batch Processing Systems

Polo et al. [97] presented an adaptive estimator for job completion time and a task scheduler that can be used for adjusting the number of allocated task slots for different MapReduce jobs to meet their completion time goals. The key hypothesis is that the execution time e_i of any task i not yet started will equal the mean execution time of the tasks that have completed until then. With e_i, it is also feasible to estimate the remaining execution time r_j of a currently running task j. Given all e_i and r_j estimates for a job as well as a completion time goal, the scheduler can calculate how many slots to concurrently allocate to the job (over time) for the job to meet its deadline. However, the estimator only monitors the progress of map tasks and makes the (often incorrect) assumption that map tasks have the same computational cost as reduce tasks.

MROnline [78] is a system that supports dynamic parameter tuning at the level of tasks for improving the performance of Hadoop jobs. In particular, it enables having different configurations for each task and can even change some task settings on the fly. MROnline differentiates between aggressive and conservative tuning. In the former, MROnline tries out multiple configurations within a single job execution and collects runtime statistics such as MapReduce counters as well as CPU, memory, and I/O utilization. The collected information along with a *gray-box hill-climbing algorithm* determine the parameter settings to try out in the next run, and the process continues until it converges to near-optimal settings. With conservative tuning, the goal is to boost job performance within a single execution. In this case, the job uses default parameter values during the first wave as it collects profiling information, while subsequent waves use tuned parameters.

Ant [29] is an adaptive self-tuning approach for finding near-optimal settings for individual tasks executed in heterogeneous environments. Initially, *Ant* categorizes nodes into clusters based

on their hardware capabilities and then applies the self-tuning approach on each cluster independently. Upon a MapReduce job submission, *Ant* will execute the first task wave with randomly selected configurations and collect runtime statistics. When the wave completes, Ant will adaptively alter the settings of the next task wave according to the best-performing tasks for each cluster. To increase the rate of tuning and avoid local optima, Ant employs a *Genetic Algorithm* for re-configuring the task parameters. Since it might take several rounds of tuning for task settings to converge, Ant performs well when jobs consist of many map tasks to be executed in multiple waves.

JellyFish [36] is another online performance tuning system that tries out different configuration settings per task until it converges to the desired settings. The overall process is similar to MROnline and Ant: Jellyfish runs each task with different parameter settings, gathers runtime information, executes the next task wave with new configuration settings, and reiterates. The two main differences to the previous approaches are that Jellyfish (1) reduces the dimensionality of search space by dividing the parameters into two groups, one for map- and one for reduce-relevant parameters; and (2) uses a *model-based hill-climbing algorithm* that takes into account interplay between different Hadoop components and the effect of parameter values.

KERMIT [45] is an online tuning system that optimizes memory and CPU assignment to individual containers by interpreting the performance of containers in terms of response time in Apache YARN. As *KERMIT* works at the level of *YARN*, it can perform performance optimization of both Hadoop MapReduce jobs and Spark applications. In particular, *YARN* intercepts resource demands coming from other platforms (e.g., MapReduce, Spark) and modifies memory sizes and CPU cores allocated to containers. KERMIT operates in either an "observe" mode or a "search" mode. In the first mode, it simply observes the effects of memory and CPU settings on the container performance and tries to discover whether a notable change in container performance has happened. If it has, then it switches to the search mode, during which KERMIT will randomly change either memory or CPU value slightly in a configurable range. By adjusting these settings, KERMIT affects container density, which affects the number of concurrent tasks executing on the cluster.

8.2 Stream Processing Systems

T-Storm [125] is a new stream data processing system based on (and backwards compatible to) Storm. *T-Storm* improves the throughput by proposing a new traffic-aware scheduler, which aims at minimizing the traffic between nodes and processes. The scheduler will occasionally determine the number of Workers to use for each topology and then assign/re-assign tasks dynamically. Also, *T-Storm* introduces a consolidation approach for using as few worker nodes as possible.

Das et al. [32] propose an adaptive algorithm for automatically determining the most suitable batch size in Spark Streaming as the environment varies. The algorithm is based on Fixed-Point Iteration, a numerical optimization technique that can continuously learn from previous batch statistics and provide low latency. The key intuition is to ensure that the batch processing time is less than the batch interval (i.e., size) so that a batch finishes processing before the next one arrives. This online algorithm is lightweight and requires no tuning on specific workloads.

DRS [43] is a dynamic resource scheduler for cloud-based data stream processing. The goal of *DRS* is to dynamically assign resources (i.e., workers and parallelism hint for each operator) to workloads in a cost-effective way, while the deadline requirements are met. Specifically, *DRS* employs performance models based on queueing theory for predicting the required resources for a given workload. Next, *DRS* uses a greedy algorithm for searching for an optimal scheduling plan after mapping the problem into a minimization problem.

Drizzle [112] is a system that aims to improve fault tolerance and adaptability for stream processing by decoupling the coordination interval from the processing interval in Spark Streaming.

Table 9. Strengths and Weaknesses of the Various Approaches for Automatic Parameter Tuning

Approach	Strengths	Weaknesses
Rule-based	• Do not require extra specialized software • Some parameters may be easy to adjust	• Time-consuming and labor-intensive process • Requires in-depth knowledge of system internals • Higher risk of performance degradation
Cost Modeling	• Very efficient for predicting performance • Good accuracy in many (basic) scenarios	• Hard to capture complexity of system internals & pluggable components (e.g., schedulers) • Models often based on simplified assumptions • Not effective on heterogeneous clusters
Simulation-based	• High accuracy in simulating dynamic system behaviors • Efficient for predicting fine-grained performance	• Hard to comprehensively simulate complex internal dynamics • Unable to capture dynamic cluster utilization • Not very efficient for finding optimal settings
Experiment-driven	• Find good settings based on real test runs on real systems • Work across different system versions and hardware	• Very time-consuming as they require multiple actual runs • Not cost effective for ad hoc applications
Machine Learning	• Ability to capture complex system dynamics • Independence from system internals and hardware • Learning based on real observations of system performance	• Require large training sets, which are expensive to collect • Training from history logs leads to data under-fitting • Typically low accuracy for unseen applications • Hard to choose the proper model
Adaptive	• Find good settings based on real test runs on real systems • Can adjust to dynamic runtime status • Work well for ad hoc applications	• Only apply to long-running applications • Inappropriate configuration can cause issues (e.g., stragglers) • Neglect efficient resource utilization in the whole system

Specifically, *Drizzle* first introduces group scheduling by arranging batches concurrently to mitigate the centralized scheduling bottleneck. Next, *Drizzle* employs pre-scheduling by rearranging the task queue based on task dependencies to alleviate the issue of straggler tasks for data dependency. Last, to achieve higher throughput, *Drizzle* further implements various optimization approaches both within and across batches.

Petrov et al. [96] propose a performance model for stream data processing to adaptively assign optimal resources (i.e., number of worker nodes and Executors) to workloads. The framework collects various statistics and system utilization metrics and then uses the models for deciding when and how to scale the current application to maximize throughput. The authors implemented the auto-scaling techniques in Spark Streaming over Amazon Web Services.

8.3 Discussion

An adaptive approach aims to dynamically adjust parameters and/or resources to ensure fault tolerance and required performance on the fly, possibly with the environment or workloads rapidly changing. As shown in Table 8, most adaptive approaches utilize some form of performance modeling for making predictions along with a search or scheduling algorithm for making tuning decisions. The methods strategically observe workload changes and adapt the resources and configuration in real-time. Such approaches work well for ad hoc and long-running (batch or streaming) workloads. However, dynamically shifting configurations in an online system may cause various performance or stability issues such as introducing stragglers or negatively affecting other concurrent workloads.

9 CONCLUSIONS AND OPEN RESEARCH CHALLENGES

Despite many differences in architecture and system internals among big data analytics systems, the key challenges related to automatic parameter tuning are the same: There is a vast number of configuration parameters that can impact system performance in non-obvious and complicated ways. To make matters worse, the performance implications vary across different applications running on clusters containing various types of CPUs, memory, storage media, and network software stack. The performance of data processing applications is further influenced by the large sizes and unstructured nature of input data, as well as variations in task execution times.

This survey presents a comprehensive study of several past approaches attempting to address the aforementioned challenges for either accurately predicting the application performance under different parameter settings or finding near-optimal parameter settings for various scenarios. The approaches are divided into six broad categories: *rule-based*, *cost modeling*, *simulation-based*, *experiment-driven*, *machine learning*, and *adaptive tuning*. Each of the six approaches excels in one or more aspects, having its unique application scenarios. Beginning with no knowledge of parameter tuning, an experiment-driven method is the most approachable one. As more experience is gained, rules or cost functions can be built to efficiently improve system performance. As the complexity of systems increases, the simulation of small components helps to increase the understanding of the system characteristics. When trying to tune very complex systems and applications with a large parameter space, machine learning methods with appropriate training data can be useful, as they typically ignore system internals. Finally, for ad hoc and long-running workloads, an adaptive approach is the best option. The strengths and weaknesses of each approach were discussed in each section and are summarized in Table 9.

As big data analytics systems increase in size and complexity, there is a growing need for automatically ensuring good and robust system performance that can cope with the ever-increasing data production rates, thus introducing more open challenges to researchers:

- **Heterogeneity:** As organizations often own multiple generations of hardware and data centers are starting to consolidate servers, heterogeneous environments are becoming common in practice. Computing-wise, nodes can have CPUs with different capacities and number of cores, while storage-wise, nodes can have multiple hard drives, SSDs, and memory of various sizes. This heterogeneity makes performance-based parameter tuning, resource allocation, and workload scheduling extremely important and challenging at the same time [69].
- **Cloud environment:** The proliferation of the Cloud and new trends such as containerization and virtualization are introducing new challenges related to multi-tenancy, overheads, and performance interactions. New Platform-as-a-Service and Software-as-a-Service offerings give rise to additional challenges in making proper resource provisioning and performance optimization decisions that can scale up to thousands of nodes, while taking economic factors (e.g., References [46, 113]) into account.
- **Real-time analytics:** The latest trend in big data analytics is to develop real-time big data pipelines [19] to enable organizations to make decisions on the fly. New challenges arise in tuning such an environment, since they are typically composed of multiple big data systems that simultaneously process data. For example, real-time analytics may use the HDFS file system, Kafka, and Spark MLlib for storing, messaging, and analyzing data, respectively.

An in-depth understanding of the existing approaches—provided by this survey—is crucial for addressing these new challenges in an effective and efficient way towards the ultimate goal of developing truly self-tuning and self-configuring systems.

REFERENCES

[1] Sean T. Allen, Matthew Jankowski, and Peter Pathirana. 2015. *Storm Applied: Strategies for Real-time Event Processing*. Manning Publications Co.

[2] AMD Hadoop Tuning. 2012. *AMD Hadoop Performance Tuning Guide*. Retrieved from https://developer.amd.com/wordpress/media/2012/10/Hadoop_Tuning_Guide-Version5.pdf.

[3] Apache Flink. 2019. *Apache Flink*. Retrieved from https://flink.apache.org/.

[4] Apache Hadoop. 2019. *Apache Hadoop*. Retrieved from https://hadoop.apache.org/.

[5] Apache Samza. 2019. *Apache Samza*. Retrieved from http://samza.apache.org/.

[6] ApacheSpark. 2019. *Apache Spark*. Retrieved from https://spark.apache.org/.

[7] Apache Spark Streaming. 2019. *Apache Spark Streaming*. Retrieved from https://spark.apache.org/streaming/.

[8] Apache Spark Tuning. 2017. *Apache Spark Tuning - DZone*. Retrieved from https://dzone.com/articles/apache-spark-performance-tuning-degree-of-parallel.

[9] Apache Spark Tuning Course. 2018. *Apache Spark Tuning and Best Practices*. Retrieved from https://databricks.com/training-overview/instructor-led-training/courses/apache-spark-tuning-and-best-practices.

[10] Apache Spark Tuning Guide. 2019. *Apache Spark Tuning Guide*. Retrieved from https://spark.apache.org/docs/latest/tuning.html.

[11] Apache Storm. 2019. *Apache Storm*. Retrieved from https://storm.apache.org/.

[12] Apache Storm Performance Tuning. 2019. *Apache Storm Performance Tuning*. Retrieved from https://storm.apache.org/releases/current/Performance.html.

[13] Apache Storm Trident. 2019. *Apache Storm Trident*. Retrieved from http://storm.apache.org/releases/current/Trident-tutorial.html.

[14] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng et al. 2015. Spark SQL: Relational data processing in Spark. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'15)*. ACM, 1383–1394.

[15] Shivnath Babu. 2010. Towards automatic optimization of MapReduce programs. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM, 137–142.

[16] Shivnath Babu and Herodotos Herodotou. 2013. Massively parallel databases and MapReduce systems. *Found. Trends® Datab.* 5, 1 (2013), 1–104.

[17] Manu Bansal, Eyal Cidon, Arjun Balasingam, Aditya Gudipati, Christos Kozyrakis, and Sachin Katti. 2018. Trevor: Automatic configuration and scaling of stream processing pipelines. *CoRR* abs/1812.09442 (2018).

[18] Liang Bao, Xin Liu, and Weizhao Chen. 2018. Learning-based automatic parameter tuning for big data analytics frameworks. In *Proceedings of the IEEE International Conference on Big Data*. IEEE, 181–190.

[19] Mike Barlow. 2013. *Real-time Big Data Analytics: Emerging Architecture*. O'Reilly Media, Inc.

[20] Ivan Bedini, Sherif Sakr, Bart Theeten, Alessandra Sala, and Peter Cogan. 2013. Modeling performance of a parallel streaming engine: Bridging theory and costs. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM, 173–184.

[21] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. 2016. RFHOC: A random-forest approach to auto-tuning Hadoop's configuration. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 27, 5 (2016), 1470–1483.

[22] Muhammad Bilal and Marco Canini. 2017. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC'17)*. ACM, 189–200.

[23] BTrace. 2018. *BTrace: A Dynamic Instrumentation Tool for Java*. Retrieved from https://github.com/btraceio/btrace.

[24] Rajkumar Buyya and Manzur Murshed. 2002. GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurr. Comput. Pract. Exper.* 14, 13–15 (2002), 1175–1220.

[25] Chi-Ou Chen, Ye-Qi Zhuo, Chao-Chun Yeh, Che-Min Lin, and Shih-Wei Liao. 2015. Machine learning-based configuration parameter tuning on Hadoop system. In *Proceedings of the IEEE International Congress on Big Data*. IEEE, 386–392.

[26] Keke Chen, James Powers, Shumin Guo, and Fengguang Tian. 2014. CRESP: Towards optimal resource provisioning for MapReduce computing in public clouds. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014), 1403–1412.

[27] Yuxing Chen, Peter Goetsch, Mohammad A. Hoque, Jiaheng Lu, and Sasu Tarkoma. 2019. d-Simplexed: Adaptive Delaunay triangulation for performance modeling and prediction on big data analytics. *IEEE Trans. Big Data* (2019). https://ieeexplore.ieee.org/document/8878273.

[28] Yuxing Chen, Jiaheng Lu, Chen Chen, Mohammad Hoque, and Sasu Tarkoma. 2019. Cost-effective resource provisioning for Spark workloads. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM'19)*. ACM, 2477–2480.

[29] Dazhao Cheng, Jia Rao, Yanfei Guo, and Xiaobo Zhou. 2014. Improving MapReduce performance in heterogeneous environments with adaptive task tuning. In *Proceedings of the 15th International Middleware Conference*. ACM, 97–108.

[30] ClouderaSparkTuning. 2018. *Cloudera Performance Management - Tuning Spark Applications*. Retrieved from https://www.cloudera.com/documentation/enterprise/5-9-x/topics/admin_spark_tuning.html.

[31] ClouderaYarnTuning. 2018. *Cloudera Performance Management - Tuning YARN*. Retrieved from https://www.cloudera.com/documentation/enterprise/5-8-x/topics/cdh_ig_yarn_tuning.html.

[32] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)*. ACM, 16:1–16:13.

[33] Databricks. 2019. *Databricks*. Retrieved from https://sparkhub.databricks.com/.

[34] Miyuru Dayarathna and Srinath Perera. 2018. Recent advancements in event processing. *Comput. Surv.* 51, 2 (2018), 33.

[35] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[36] Xiaoan Ding, Yi Liu, and Depei Qian. 2015. Jellyfish: Online performance tuning with adaptive configuration and elastic container in Hadoop YARN. In *Proceedings of the 21st International Conference on Parallel and Distributed Systems*. IEEE, 831–836.

[37] Shlomi Dolev, Patricia Florissi, Ehud Gudes, Shantanu Sharma, and Ido Singer. 2017. A survey on geographically distributed big-data processing using MapReduce. *IEEE Trans. Big Data* 5, 1 (2017), 60–80.

[38] Christos Doulkeridis and Kjetil Nørvåg. 2014. A survey of large-scale analytical query processing in MapReduce. *VLDB J.* 23, 3 (June 2014), 355–380. DOI: https://doi.org/10.1007/s00778-013-0319-9

[39] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with iTuned. *PVLDB* 2, 1 (2009), 1246–1257.

[40] Mostafa Ead, Herodotos Herodotou, Ashraf Aboulnaga, and Shivnath Babu. 2014. PStorM: Profile storage and matching for feedback-based tuning of MapReduce jobs. In *Proceedings of the 17th International Conference on Extending Database Technology (EDBT'14)*. 1–12.

[41] Lorenz Fischer, Shen Gao, and Abraham Bernstein. 2015. Machines tuning machines: Configuring distributed stream processors with Bayesian optimization. In *Proceedings of the International Conference on Cluster Computing (CLUSTER'15)*. IEEE, 22–31.

[42] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self regulating stream processing in Heron. *PVLDB* 10, 12 (2017), 1825–1836.

[43] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2015. DRS: Dynamic resource scheduling for real-time analytics over fast streams. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'15)*. IEEE, 411–420.

[44] Jyoti V. Gautam, Harshadkumar B. Prajapati, Vipul K. Dabhi, and Sanjay Chaudhary. 2015. A survey on job scheduling algorithms in big data processing. In *Proceedings of the International Conference on Electrical, Computer and Communication Technologies*. IEEE, 1–11.

[45] Mikhail Genkin, Frank Dehne et al. 2016. Automatic, on-line tuning of YARN container memory and CPU parameters. In *Proceedings of the International Conference on High Performance Computing and Communications (HPCC'16)*. IEEE, 317–324.

[46] Anastasios Gounaris, Georgia Kougka, Ruben Tous, Carlos Tripiana Montes, and Jordi Torres. 2017. Dynamic configuration of partitioning in Spark applications. *IEEE Trans. Parallel Distrib. Syst.* 28, 7 (2017), 1891–1904.

[47] Anastasios Gounaris and Jordi Torres. 2017. A methodology for Spark parameter tuning. *Big Data Res.* 11 (Mar. 2017), 22–32.

[48] HadoopClusterSetup. 2019. *Hadoop Cluster Setup*. Retrieved from https://hadoop.apache.org/docs/r1.2.1/cluster_setup.html.

[49] HadoopPerfUI. 2011. *Hadoop Perf Monitoring UI*. Retrieved from http://code.google.com/p/hadoop-toolkit/wiki/HadoopPerformanceMonitoring.

[50] HadoopTuning. 2015. *Hadoop Performance Tuning Tutorial*. Retrieved from http://hadooptutorial.info/hadoop-performance-tuning/.

[51] HadoopTutorial. 2018. *Hadoop MapReduce Tutorial*. Retrieved from https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[52] HadoopVaidya. 2011. *Hadoop Vaidya*. Retrieved from http://hadoop.apache.org/mapreduce/docs/r0.21.0/vaidya.html.

[53] Suhel Hammoud, Maozhen Li, Yang Liu, Nasullah Khalid Alham, and Zelong Liu. 2010. MRSim: A discrete event based MapReduce simulator. In *Proceedings of the 7th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD'10)*, Vol. 6. IEEE, 2993–2997.

[54] Dominique Heger. 2013. Hadoop Performance Tuning—A Pragmatic & Iterative Approach. Retrieved from https://www.cmg.org/wp-content/uploads/2013/04/m_97_3.pdf.
[55] Álvaro Brandón Hernández, María S. Perez, Smrati Gupta, and Victor Muntés-Mulero. 2017. Using machine learning to optimize parallelism in big data applications. *Fut. Gen. Comput. Syst.* 86 (2018), 1076–1092. https://www.sciencedirect.com/science/article/abs/pii/S0167739X17314668?via%3Dihub.
[56] Herodotos Herodotou. 2011. Hadoop performance models. *CoRR* abs/1106.0940 (2011).
[57] Herodotos Herodotou. 2012. *Automatic Tuning of Data-intensive Analytical Workloads*. Ph.D. Dissertation. Duke University.
[58] Herodotos Herodotou and Shivnath Babu. 2011. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *PVLDB* 4, 11 (2011), 1111–1122.
[59] Herodotos Herodotou and Shivnath Babu. 2013. A what-if engine for cost-based MapReduce optimization. *IEEE Data Eng. Bull.* 36, 1 (2013), 5–14.
[60] Herodotos Herodotou, Fei Dong, and Shivnath Babu. 2011. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC'11)*.
[61] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A self-tuning system for big data analytics. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR'11)*. 261–272.
[62] Wilson A. Higashino, Miriam A. M. Capretz, and Luiz F. Bittencourt. 2016. CEPSim: Modelling and simulation of complex event processing systems in cloud environments. *Fut. Gen. Comput. Syst.* 65 (2016), 122–139.
[63] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *Comput. Surv.* 46, 4 (2014), 46.
[64] Fred Howell and Ross McNab. 1998. SimJava: A discrete event simulation library for Java. *Simul. Series* 30 (1998), 51–56.
[65] Markus C. Huebscher and Julie A. McCann. 2008. A survey of autonomic computing—degrees, models, and applications. *Comput. Surv.* 40, 3 (2008), 7:1–7:28.
[66] Pooyan Jamshidi and Giuliano Casale. 2016. An uncertainty-aware approach to optimal configuration of stream processing systems. In *Proceedings of the IEEE/ACM International Symposium on Modeling, Analysis, and Simulation on Computer and Telecommunication Systems (MASCOTS'16)*. IEEE, 39–48.
[67] Zhen Jia, Chao Xue, Guancheng Chen, Jianfeng Zhan, Lixin Zhang, Yonghua Lin, and Peter Hofstee. 2016. Auto-tuning Spark big data workloads on POWER8: Prediction-based dynamic SMT threading. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT'16)*. IEEE, 387–400.
[68] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. 2010. The performance of MapReduce: An in-depth study. *PVLDB* 3, 1–2 (2010), 472–483.
[69] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware distributed parameter servers. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'17)*. ACM, 463–478.
[70] Selvi Kadirvel and José A. B. Fortes. 2012. Grey-box approach for performance prediction in MapReduce based platforms. In *Proceedings of the 21st International Conference on Computer Communications and Networks (ICCCN'12)*. IEEE, 1–9.
[71] Faria Kalim, Thomas Cooper, Huijun Wu, Yao Li, Ning Wang, et al. 2019. Caladrius: A performance modelling service for distributed stream processing systems. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE'19)*. IEEE, 1886–1897.
[72] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. 2010. An analysis of traces from a production MapReduce cluster. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 94–103.
[73] Mukhtaj Khan, Yong Jin, Maozhen Li, Yang Xiang, and Changjun Jiang. 2016. Hadoop performance modeling for job estimation and resource provisioning. *IEEE Trans. Parallel Distrib. Syst.* 27, 2 (2016), 441–454.
[74] Johannes Kroß and Helmut Krcmar. 2017. Model-based performance evaluation of batch and stream applications for big data. In *Proceedings of the IEEE/ACM International Symposium on Modeling, Analysis, and Simulation on Computer and Telecommunication Systems (MASCOTS'17)*. IEEE Computer Society, 80–86.
[75] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, et al. 2015. Twitter Heron: Stream processing at scale. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'15)*. ACM, 239–250.
[76] Palden Lama and Xiaobo Zhou. 2012. AROMA: Automated resource allocation and configuration of MapReduce environment in the cloud. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC'12)*. ACM, 63–72.
[77] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. 2012. Parallel data processing with MapReduce: A survey. *ACM SIGMOD Record* 40, 4 (Jan. 2012), 11–20. DOI: https://doi.org/10.1145/2094114.2094118.

[78] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, et al. 2014. MROnline: MapReduce online performance tuning. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC'14)*. ACM, 165–176.

[79] Teng Li, Jian Tang, and Jielong Xu. 2016. Performance modeling and predictive scheduling for distributed stream data processing. *IEEE Trans. Big Data* 2, 4 (2016), 353–364.

[80] Guangdeng Liao, Kushal Datta, and Theodore L. Willke. 2013. Gunther: Search-based auto-tuning of MapReduce. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'13)*. Springer, 406–419.

[81] Jia-Chun Lin, Ming-Chang Lee, Ingrid Chieh Yu, and Einar Broch Johnsen. 2018. Modeling and simulation of Spark streaming. In *Proceedings of the International Conference on Advanced Information Networking and Applications (AINA'18)*. IEEE, 407–413.

[82] Jia-Chun Lin, Ingrid Chieh Yu, Einar Broch Johnsen, and Ming-Chang Lee. 2016. ABS-YARN: A formal framework for modeling Hadoop YARN clusters. In *Proceedings of the Fundamental Approaches to Software Engineering Conference (FASE'16) (Lecture Notes in Computer Science)*, Vol. 9633. Springer, 49–65.

[83] Xuelian Lin, Zide Meng, Chuan Xu, and Meng Wang. 2012. A practical performance model for Hadoop MapReduce. In *Proceedings of the IEEE International Conference on Cluster Computing Workshops (CLUSTER WORKSHOPS'12)*. IEEE, 231–239.

[84] Chao Liu, Deze Zeng, Hong Yao, Chengyu Hu, Xuesong Yan, and Yuanyuan Fan. 2015. MR-COF: A genetic MapReduce configuration optimization framework. In *Proceedings of the International Conference on Algorithms and Architecture for Parallel Processing*. Springer, 344–357.

[85] Jun Liu, Nishkam Ravi, Srimat Chakradhar, and Mahmut Kandemir. 2012. Panacea: Towards holistic optimization of MapReduce applications. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*. ACM, 33–43.

[86] Xunyun Liu, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, et al. 2018. A stepwise auto-profiling method for performance optimization of streaming applications. *ACM Trans. Auton. Adapt. Syst.* 12, 4 (2018), 24:1–24:33.

[87] Yang Liu, Maozhen Li, Nasullah Khalid Alham, and Suhel Hammoud. 2013. HSim: A MapReduce simulator in enabling cloud computing. *Fut. Gen. Comput. Syst.* 29, 1 (2013), 300–308.

[88] Jiaheng Lu, Yuxing Chen, Herodotos Herodotou, and Shivnath Babu. 2019. Speedup your analytics: Automatic parameter tuning for databases and big data systems. *PVLDB* 12, 12 (2019), 1970–1973.

[89] Michael Malak and Robin East. 2016. *Spark GraphX in Action*. Manning Publications Co.

[90] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. MLlib: Machine learning in Apache Spark. *J. Mach. Learn. Res.* 17, 1 (2016), 1235–1241.

[91] Matt Morgan. 2015. *Ensuring the Best Performance from Your Hadoop Clusters, Proactively*. Retrieved from https://hortonworks.com/blog/ensuring-the-best-performance-from-your-hadoop-clusters-proactively/.

[92] Mumak. 2010. *Mumak: Map-Reduce Simulator*. Retrieved from https://issues.apache.org/jira/browse/MAPREDUCE-728.

[93] Zubair Nabi, Eric Bouillet, Andrew Bainbridge, and Chris Thomas. 2014. Of streams and storms. *IBM White Paper* (2014), 1–31.

[94] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, et al. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'09)*. ACM, 165–178.

[95] Panagiotis Petridis, Anastasios Gounaris, and Jordi Torres. 2016. Spark parameter tuning via trial-and-error. In *Proceedings of the INNS Conference on Big Data*. Springer, 226–237.

[96] Max Petrov, Nikolay Butakov, Denis Nasonov, and Mikhail Melnik. 2018. Adaptive performance model for dynamic scaling apache spark streaming. *Procedia Comput. Sci.* 136 (2018), 109–117.

[97] Jorda Polo, David Carrera, Yolanda Becerra, Jordi Torres, Eduard Ayguadé, Malgorzata Steinder, and Ian Whalley. 2010. Performance-driven task co-scheduling for MapReduce environments. In *Proceedings of the IEEE Network Operations and Management Symposium (NOMS'10)*. IEEE, 373–380.

[98] José Ignacio Requeno, José Merseguer, and Simona Bernardi. 2017. Performance analysis of Apache Storm applications using stochastic petri nets. In *Proceedings of the International Conference on Information Reuse and Integration (IRI'17)*. IEEE, 411–418.

[99] Henriette Röger and Ruben Mayer. 2019. A comprehensive survey on parallelization and elasticity in stream processing. *Comput. Surv.* 52, 2 (2019), 36.

[100] Rumen 2009. *Rumen: A Tool to Extract Job Characterization Data from Job Tracker Logs*. Retrieved from https://issues.apache.org/jira/browse/MAPREDUCE-751.

[101] Matthias J. Sax, Malu Castellanos, Qiming Chen, and Meichun Hsu. 2013. Performance optimization for distributed intra-node-parallel streaming systems. In *Proceedings of the 29th International Conference on Data Engineering Workshops (ICDEW'13)*. IEEE, 62–69.

[102] Juwei Shi, Jia Zou, Jiaheng Lu, Zhao Cao, Shiqiang Li, and Chen Wang. 2014. MRTuner: A toolkit to enable holistic optimization for mapreduce jobs. *PVLDB* 7, 13 (2014), 1319–1330.

[103] Rekha Singhal and Praveen Singh. 2017. Performance assurance model for applications on SPARK platform. In *Proceedings of the Technology Conference on Performance Evaluation and Benchmarking (TPCTC'17)*. Springer, 131–146.

[104] SparkCoreParameter 2019. *Spark Core Parameters*. Retrieved from https://spark.apache.org/docs/latest/configuration.html.

[105] Nicoleta Tantalaki, Stavros Souravlas, and Manos Roumeliotis. 2019. A review on big data real-time stream processing and its scheduling techniques. *Int. J. Parallel Emerg. Distrib. Syst.* (2019), 1–31. https://www.tandfonline.com/doi/abs/10.1080/17445760.2019.1585848.

[106] Fei Teng, Lei Yu, and Frederic Magoulès. 2011. SimMapReduce: A simulator for modeling MapReduce framework. In *Proceedings of the 5th FTRA International Conference on Multimedia and Ubiquitous Engineering (MUE'11)*. IEEE, 277–282.

[107] TheNS2. 2011. *The Network Simulator - ns-2*. Retrieved from https://www.isi.edu/nsnam/ns/.

[108] Michael Trotter, Guyue Liu, and Timothy Wood. 2017. Into the storm: Descrying optimal configurations using genetic algorithms and Bayesian optimization. In *Proceedings of the IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W'17)*. IEEE Computer Society, 175–180.

[109] Michael Trotter, Timothy Wood, and Jinho Hwang. 2019. Forecasting a storm: Divining optimal configurations using genetic algorithms and supervised learning. In *Proceedings of the International Conference on Autonomic Computing (ICAC'19)*. IEEE, 136–146.

[110] Luis M. Vaquero and Félix Cuadrado. 2018. Auto-tuning distributed stream processing systems using reinforcement learning. *CoRR* abs/1809.05495 (2018).

[111] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, et al. 2013. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC'13)*. ACM, 5.

[112] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'17)*. ACM, 374–389.

[113] Shivaram Venkataraman, Zongheng Yang, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, 363–378.

[114] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. 2011. ARIA: Automatic resource inference and allocation for MapReduce environments. In *Proceedings of the 8th International Conference on Autonomic Computing (ICAC'11)*. ACM, 235–244. DOI: https://doi.org/10.1145/1998582.1998637

[115] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. 2011. Play it again, SimMR! In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'11)*. IEEE, 253–261.

[116] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. 2011. Resource provisioning framework for MapReduce jobs with performance goals. In *Proceedings of the ACM/IFIP/USENIX 12th International Middleware Conference*. Springer, 165–186.

[117] Chunkai Wang, Xiaofeng Meng, Qi Guo, Zujian Weng, and Chen Yang. 2017. Automating characterization deployment in distributed data stream management systems. *IEEE Trans. Knowl. Data Eng.* 29, 12 (2017), 2669–2681.

[118] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta. 2009. A simulation approach to evaluating design decisions in MapReduce setups. In *Proceedings of the IEEE/ACM International Symposium on Modeling, Analysis, and Simulation on Computer and Telecommunication Systems (MASCOTS'09)*. IEEE, 1–11.

[119] Guolu Wang, Jungang Xu, and Ben He. 2016. A novel method for tuning configuration parameters of Spark based on machine learning. In *Proceedings of the International Conference on High Performance Computing and Communications (HPCC'16)*. IEEE, 586–593.

[120] Kewen Wang and Mohammad Maifi Hasan Khan. 2015. Performance prediction for Apache Spark platform. In *Proceedings of the International Conference on High Performance Computing and Communications (HPCC'15)*. IEEE, 166–173.

[121] Kewen Wang, Xuelian Lin, and Wenzhong Tang. 2012. Predator—an experience guided configuration optimizer for Hadoop MapReduce. In *Proceedings of the IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom'12)*. IEEE, 419–426.

[122] Thomas Weise. 2009. *Global Optimization Algorithms—Theory and Application*. Self-published. http://www.it-weise.de/projects/book.pdf.

[123] Tom White. 2012. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc.

[124] Dili Wu and Aniruddha Gokhale. 2013. A self-tuning system based on application profiling and performance analysis for optimizing Hadoop MapReduce cluster configuration. In *Proceedings of the 20th International Conference on High Performance Computing (HiPC'13)*. IEEE, 89–98.

[125] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. 2014. T-Storm: Traffic-aware online scheduling in Storm. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'14)*. IEEE, 535–544.

[126] Hailong Yang, Zhongzhi Luan, Wenjun Li, Depei Qian, and Gang Guan. 2012. Statistics-based workload modeling for MapReduce. In *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2043–2051.

[127] Tao Ye and Shivkumar Kalyanaraman. 2003. A recursive random search algorithm for large-scale network parameter configuration. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*. ACM, 196–205.

[128] Nezih Yigitbasi, Theodore L. Willke, Guangdeng Liao, and Dick Epema. 2013. Towards machine learning–based auto-tuning of MapReduce. In *Proceedings of the IEEE/ACM International Symposium on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems (MASCOTS'13)*. IEEE, 11–20.

[129] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM, 564–577.

[130] Nikos Zacheilas, Vana Kalogeraki, Nikolaos Zygouras, Nikolaos Panagiotou, and Dimitrios Gunopulos. 2015. Elastic complex event processing exploiting prediction. In *Proceedings of the IEEE International Conference on Big Data (Big Data'15)*. IEEE Computer Society, 213–222.

[131] Nikos Zacheilas, Stathis Maroulis, and Vana Kalogeraki. 2017. Dione: Profiling Spark applications exploiting graph similarity. In *Proceedings of the IEEE International Conference on Big Data (BigData'17)*. IEEE, 389–394.

[132] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, 2–14. Retrieved from http://dl.acm.org/citation.cfm?id=2228298.2228301.

[133] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. 2014. Parameterizable benchmarking framework for designing a MapReduce performance model. *Concurr. Comput. Pract. Exper.* 26, 12 (2014), 2005–2026.

[134] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, et al. 2017. BestConfig: Tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC'17)*. ACM, 338–350.

Received March 2019; revised December 2019; accepted January 2020